

目 录

第 1 章 数据通信基础	1
1.1 数据通信的基本概念	1
1.1.1 同步通信和异步通信	1
1.1.2 波特率与数据传输率	2
1.2 异步串行通信协议	3
1.2.1 异步串行通信协议	3
1.2.2 定制通信协议	4
1.3 DCE 设备——Modem	5
1.3.1 Modem 的基本原理	5
1.3.2 Modem 的传输速率	6
1.3.3 Modem 的类型	7
1.3.4 Modem 的通信协议体系	7
1.3.5 Modem 的安装与使用	11
1.3.6 外置式调制解调器的指示灯	11
1.3.7 Modem 技术的新发展	12
本章小结	13
第 2 章 通用串行通信标准和通用 Modem 命令	14
2.1 RS-232C 标准	14
2.1.1 信号连接	15
2.1.2 握手	16
2.1.3 微机的 RS-232C 接口	18
2.2 通用 Modem 命令	20
2.2.1 Modem 状态	20
2.2.2 AT 命令	21
2.2.3 S 寄存器	32
2.2.4 Modem 返回信息码	35
2.3 通用异步接收发送器 UART 概述	36
本章小结	37
第 3 章 嵌入式汇编语言开发通信程序	38
3.1 Delphi 中的嵌入式汇编语言	38
3.1.1 汇编语句的基础知识	38

3.1.2 表达式	42
3.1.3 汇编程序过程和函数	48
3.2 嵌入式汇编的通信编程例子	49
3.2.1 在 Delphi 中对端口的直接操作	49
3.2.2 行间汇编接收下位机传来的数据的简单例子	49
3.2.3 用于串行通信的 Delphi DLL 程序	50
3.2.4 直接操作端口的 Delphi 单元	53
本章小结	55
第 4 章 MSComm 控件应用	56
4.1 MSComm 控件	56
4.1.1 MSComm 控件方法	56
4.1.2 MSComm 控件属性	57
4.1.3 MSComm 控件事件的介绍	69
4.2 MSComm 控件的错误消息	70
4.3 用 MSComm 控件编程实例	70
4.3.1 简单的 MSComm 程序分析	70
4.3.2 复杂的 MSComm 程序实例和分析	74
4.4 使用技巧	94
本章小结	99
第 5 章 线程开发	100
5.1 线程简介	100
5.1.1 进程和线程	101
5.1.2 线程的同步	103
5.1.3 线程的优先级	104
5.1.4 线程实例	105
5.2 TThread 对象	107
5.2.1 TThread 对象	107
5.2.2 TThread 实例	110
本章小结	116
第 6 章 Windows API 通信编程	117
6.1 串口通信 API 函数	117
6.1.1 Windows 98 和 Windows 3.x 通信结构	117
6.1.2 串口通信 API 函数介绍	119
6.1.3 示例程序和分析	125
6.2 基于 Windows TAPI 通信编程	135
6.2.1 电话编程接口的简介	135
6.2.2 TAPI 主要函数和基于 TAPI 应用的基本步骤介绍	135

6.2.3 基于 TAPI 通信例子	141
本章小结	152
第 7 章 其他通信控件的使用	153
7.1 SPComm 控件的使用	153
7.1.1 SPComm 的主要属性、方法和事件	154
7.1.2 SPComm 控件的串口通信例子	155
7.2 Turbopower 的 APRO 组件	158
7.2.1 TApdComPort 控件	159
7.2.2 TApdRasDialer 控件	163
7.2.3 TApdRasStatus 控件	165
7.2.4 TApdSModem 控件	165
7.2.5 TApdModem 控件	167
7.2.6 TApdSLController 控件	169
7.2.7 TApdStatusLight 控件	169
7.2.8 TApdProtocol 控件	169
7.2.9 TApdProtocolLog 控件	170
7.2.10 TApdProtocolStatus 控件	170
7.3 Turbopower 的 APRO 2.x 组件	171
7.3.1 TApdModemDBase 控件	171
7.3.2 TApdModemDialer 控件	172
7.3.3 TAdTerminal 控件	174
7.3.4 TApdPhoneNumberSelector 控件	175
7.4 基于 APRO 组件的例子	176
本章小结	183
第 8 章 基于 MSComm 的多线程通信编程实例详解	184
8.1 系统简介	184
8.1.1 告警监测仪 (包括监测单元、调制解调器、采集器)	184
8.1.2 监控中心	185
8.2 系统规划设计	186
8.2.1 各模块说明	186
8.2.2 通信协议	188
8.2.3 通信日志设计	192
8.2.4 数据库设计	193
8.3 源程序的分析	198
8.3.1 循环冗余校验 CRC 算法源程序的分析	198
8.3.2 信息包的处理	202
8.3.3 通信线程的分析	215
8.4 异常处理在程序中的应用	244

本章小结	245
第9章 RAS 编程	246
9.1 RAS 基本知识	246
9.2 拨号网络的配置	247
9.2.1 Windows NT 4.0 拨号服务器配置	247
9.2.2 Windows 2000 远程访问服务器的配置	249
9.2.3 拨号客户端主机的配置	250
9.3 在程序中实现 RAS	250
9.3.1 RAS 的 API 函数简介	251
9.3.2 使用动态链接库实现 RAS 的函数调用	251
9.3.3 在 Delphi 程序中拨号上网	254
9.3.4 断开 Internet 连接的程序	257
9.3.5 使用拨号网络的类 Tras	262
本章小结	266
第10章 通信安全设计	267
10.1 数据加密基础知识	267
10.1.1 加密技术	267
10.1.2 数字签名 (Digital Signature)	269
10.1.3 数字信封	270
10.2 应用编程接口编程模式	271
10.3 微软信息密码系统	271
10.4 创建签名消息	273
10.4.1 CertOpenStore	274
10.4.2 CertCloseStore	274
10.4.3 CryptSignMessage	275
10.5 加密并封装一个消息	276
10.5.1 CryptMsgOpenToEncode	277
10.5.2 CryptMsgUpdate	278
10.5.3 CryptMsgGetParam	278
10.5.4 CryptMsgClose	279
10.6 解密封装的数据 (或者解封数据)	279
CryptMsgOpenToDecode	280
10.7 校验签名的消息	281
关键函数 CryptVerifyMessageSignature	282
10.8 加密算法源码分析	283
10.8.1 加密、解密函数库	284
10.8.2 Delphi 例子	303
本章小结	307

第 11 章 强大的项目管理工具 Rational Rose	308
11.1 Rose 简介	309
11.2 Rose Delphi Link 简介	310
11.2.1 RDL 的操作原则	311
11.2.2 使用 Rose Delphi Link	312
11.2.3 修改 RDL 的代码生成特性	317
11.3 UML 简介	322
11.3.1 标准建模语言 UML 的出现	322
11.3.2 标准建模语言 UML 的内容	324
11.3.3 标准建模语言 UML 的主要特点	325
11.4 Rose 在项目设计和管理中的具体应用	326
11.4.1 UML 建模过程高层视图	326
11.4.2 UML 实际建模过程	327
11.5 参考	333
本章小结	337

第 1 章 数据通信基础

本章主要内容：

- 同步通信与异步通信基本概念
- 异步串行通信协议
- Modem 的基本原理
- Modem 的类型
- Modem 的通信协议体系
- Modem 的安装与使用

本章从技术人员的角度介绍数据通信基本概念，Modem 的基本原理、类型和通信协议体系以及安装和使用 Modem 的基本知识。

1.1 数据通信的基本概念

终端设备和计算机之间的通信或计算机和计算机之间的通信，通常称为数据通信。进行数据通信的双方必须遵守相同的传输控制规程（或通信协议规程）才能协调、可靠地工作。数据通信系统的基本构成如图 1.1 所示。图中的数据通信系统中传输控制器和通信控制器主要功能就是完成数据通信的传输控制规程。数据电路和传输控制规程称为数据链路。

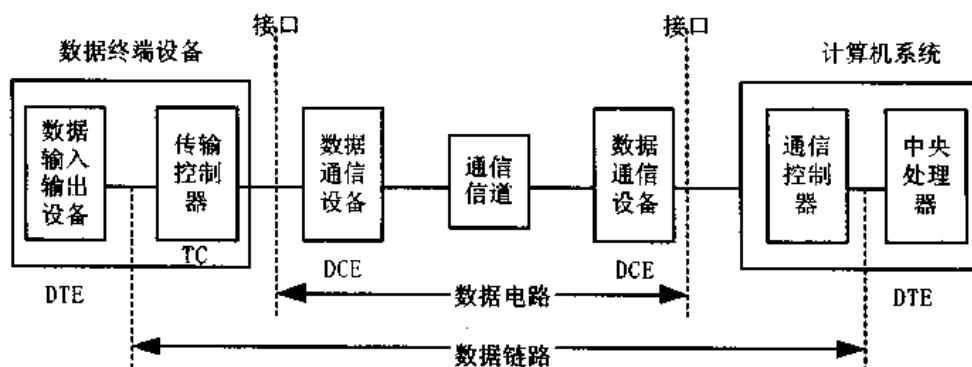


图 1.1 数据通信系统的基本构成

1.1.1 同步通信和异步通信

数据通信方式有两种：同步通信方式和异步通信方式。

1. 同步通信方式

同步通信 (Synchronous Communication) 方式要求通信双方以相同的速率进行, 而且要准确协调。它通过共享一个单个时钟或定时脉冲源保证发送方和接收方准确同步。其特点是允许发送一个字符序列, 每个字符数据位数相同, 没有起始位和停止位, 效率高。同步通信以多字节组成的数据块 (几十至几千个字节) 为单位进行传输, 并在数据块前加上标识序列组成帧 (Frame)。同步方式分字节同步和位同步两种, 通常采用后一种方式, 例如 ISO 的 HDLC 等。

2. 异步通信方式

异步通信 (Asynchronous Communication) 不要求双方同步, 收发方可采用各自的时钟源。双方都遵循异步通信协议, 以字符为数据传输单位, 发送方传送字符的时间间隔不确定。每个字符传输都以起始位开始, 以停止位结束。通信双方所指定的字符的数据位数, 奇偶校验方法和停止位数必须相同, 传输效率比同步通信方式低, 成本也低。异步通信以字符为单位, 长度为 5~8 位。

异步通信是“起——止” (Start——Stop Mode) 同步方式通信, 即在以起始位开始、停止位结束的一个字符内按约定的频率进行同步接收。各字符之间允许有间隙, 而且两个字符之间的间隔是不固定的。而在同步通信方式中, 不仅同一字符中的相邻两位间的时间间隔要相等, 而且相邻字符间的间隔也要求相等, 这就是同步与异步通信方式的主要差别。例如终端设备的输出速度不均匀时, 就要求相邻字符间的时间间隔不同, 这时就要用到异步通信方式。

1.1.2 波特率与数据传输率

通常衡量数据通信能力有以下两种方法。

1. 波特率 (又称调制速率)

波特率指单位时间内线路状态变化的次数, 反映了数据的调制信号波形变换的频繁程度, 单位是“波特” (Baud)。

2. 数据传输率

数据传输率指单位时间内传送的信息量, 以每秒内传送的二进制数据“1”和“0”的数量表示, 单位是“比特/秒” (bit/s)。

波特率与数据传输率两者相似但不等同。当采用基波传输 (零调制或空调制) 时, 且单位时间内仅调制/解调 1 个信号, 两者速率数值相同。

当采用载波传输时, 波特率与数据传输率关系如下:

$$C = B \log_2 n$$

其中: C 为数据传输率, B 为波特率。 n 为调制信号数或线路状态数 (2 的倍数)。

波特率与数据传输率的关系如图 1.2 所示。

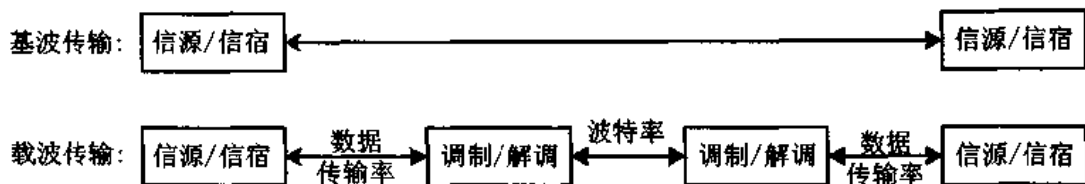


图 1.2 波特率与数据传输率关系图

对 PC 机而言，波特率与数据传输率数值相同。

1.2 异步串行通信协议

在自动化控制应用中，经常需要计算机与外围设备进行数据通信。而异步串行通信是一种常用的通信手段。下面详细介绍异步串行通信协议。

1.2.1 异步串行通信协议

通信协议也叫通信规程，是指通信双方一种格式上的约定。数据通信中，在收发器之间传送的是一组二进制的“0”、“1”位串，但它们在不同的位置可能有不同的含义，有的只是用于同步，有的代表了通信双方的地址，有的是一些控制信息，有的则是通信中真正要传输的数据，还有的是为了差错控制而附加上去的冗余位。这些都需要在通信协议中事先约定好，以形成一种收/发双方共同遵守的格式。

在逐位传送的串行通信中，接收端必须能识别每个二进制位从什么时刻开始，这就是位定时。通信中一般以若干位表示一个字符，除了位定时外，还需要在接收端能识别每个字符从哪位开始，这是字符定时。

异步串行通信时，每个字符作为一个独立的信息，可以随机出现在数据流中，即每个字符出现在数据流中的相对时间是任意的。然而，一个字符一旦开始出现，字符中各位则是以预先固定的时钟传送。因此，异步通信方式的“异步”主要体现在字符与字符之间，而同一字符内部的位与位间是同步的。为确保异步通信的正确性，必须找到一种方法，使收发双方在随机传送的字符内部实现同步。这种方法就是在字符格式中设置起始位和停止位，即在一个字符正式发送之前先发一个起始位，该字符结束时再发一个停止位。接收器检测到起始位便知道字符到达，并开始接收字符、检测到停止位则知道字符传输已结束。由于这种通信协议是靠起始位和停止位来实现字符内部同步的，所以有时也称为起止式协议。

异步通信的传输格式如图 1.3 所示。每帧信息（即每个字符）由 4 部分组成：

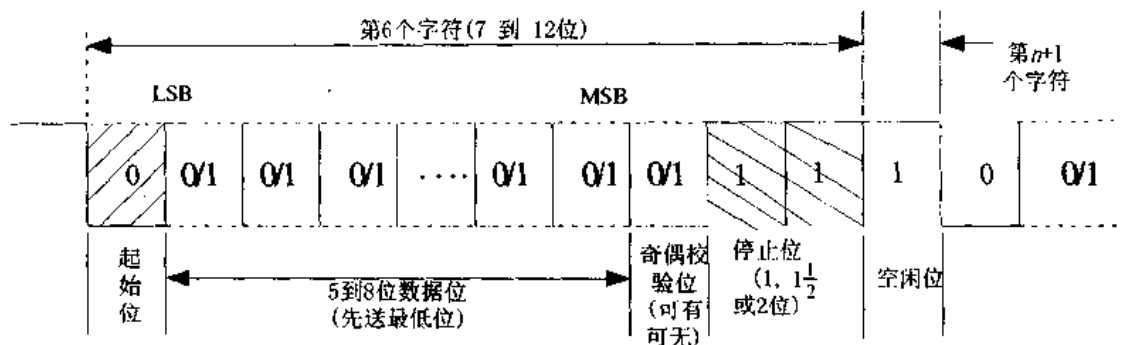


图 1.3 为异步串行通信格式

(1) 1 位起始位，规定为低电平“0”。

(2) 5~8 位数据位，它紧跟在起始位后面，是要传送的有效信息。规定从低位至高位依次传送。

(3) 0~1 位奇偶校验位。

(4) 1 位、1.5 位或 2 位停止位，规定为高电平。

异步通信格式中起始位和停止位至关重要。起始位标志每个字符的开始，通知接收器开始装配一个字符，以便和发送器取得同步。停止位标志每个字符的结束。通过起始位和停止位的巧妙结合，实现异步字符传输的同步。由于这种同步只需在一个字符期间保持，下一个字符又将包含新的起始位和停止位，所以发送器和接收器不必使用同一个时钟，只需分别使用两个频率相同的局部时钟，使它们在一个字符时间内收发双方的串行位流能保持同步，即可做到正确可靠地传送。进行异步通信时，关键是接收器必须能准确地发现每个字符开始出现的时刻，为此，协议规定起始位和停止位必须采用相反的极性，前者为低电平“0”，后者为高电平“1”。利用前一个字符的高电平停止位到后一个字符的低电平起始位的负跳变，接收器知道这是一个字符的开始，可以以此作为新字符内位检测与采样的时间基准。为了让一个字符到另一个字符的转换总是以负跳变开始，通信协议规定在字符与字符之间出现空闲状态时，空闲位一律用停止位的“1”填充。

停止位长度规定可以取 1 位、1.5 位或 2 位。一般有效数据为 5 位（称为 5 单位字符码）时停止位取 1 位或 1.5 位，其他单位的字符码停止位取 1 位或 2 位。至于有效数据位后面的奇偶校验位，协议规定可有可无。

在数据通信中，按照 ITU（International Telecommunication Union，国际电信联盟）的建议，通常将逻辑“0”称为“空号”（Space），而将逻辑“1”称为“传号”（Mark）。按这种叫法，在异步串行通信中，每个字符的传送都必须以“空号”开始，以“传号”填充空闲位。

1.2.2 自定义通信协议

在异步传输中，通信协议的数据字符格式为起始位+数据位+校验位+停止位。常用格式有：

- ☐ 1bit 起始位+8bit 数据位+1bit 停止位（无校验位）
- ☐ 1bit 起始位+7bit 数据位+1 位偶校验位+1bit 停止位

1 个字符通常用 10bit 表示。数据格式可在通信软件（如 Windows 98 中的“超级终端”）

中设置。

现在, 自动化控制系统常采用单片机作为下位机, PC 机 (或服务器) 为上位机, 二者通过 RS-232C 串行口接收或发送数据和指令。

上位机和下位机的通信协议定制在系统设计中很重要, 笔者在下面列出一个简单的自定义通信协议。系统采用 PC 机承担主控任务。单片用来机接受 PC 机指令, 并根据指令信息驱动并控制机器或发送数据给 PC 机。通信协议如下:

采用 RS-232C 串口异步通信, 1 位起始位, 8 位数据位, 无奇偶校验位, 1 位停止位, 波特率 2400bit/s。传输数据采用 ASCII 模式。指令形式采用 7 个 ASCII 串, 格式为 “\$××××××*”, 其中 “\$” 和 “*” 分别标明该指令的起始和结束, “×××××” 为指令内容。如: “\$REMOT*” 为远程控制, “\$H±××××*” 为工业机器给定命令, “\$ASKQ?*” 为主机查询命令等。单片机按接收到的指令工作。如果主控机发出错误的指令或现在正执行上一条给定命令的过程中又收到新的给定命令, 将不做任何控制, 并显示 Error 提示, 1 秒钟后自动返回。

1.3 DCE 设备——Modem

DTE (Data Terminal Equipment, 数据终端设备) 是用于发送和接收数据的设备, 可以是一台计算机, 也可以是一台只接收数据的打印机。

DCE (Data Communications Equipment, 数据通信设备) 是用来连接 DTE 与通信网络的设备, 可以是一台调制解调器, 也可以是一个简单的线路驱动器。

目前使用最广泛的数据传输信道就是模拟电话线路, 计算机所能处理的数字信号不能直接进入这样的信道, 而必须通过一个信号变换装置——Modem。Modem 用来对数据信号进行变换, 使变换后的信号可以适应信道传输的特性, 延长数据信号的传送距离。Modem 是最重要的 DCE 设备之一。

1.3.1 Modem 的基本原理

Modem 其实是 Modulator (调制器) 与 Demodulator (解调器) 的简称, 中文称为调制解调器。也有人根据 Modem 的谐音, 昵称为 “猫”。

计算机是通过 “0” 和 “1” 这两个数字来完成信息交换的, 而电话线是传送模拟信号的, 因此需要有进行模拟信号与数字信号转换的设备。调制是一个将计算机发出的二进制信号转变为可以通过普通电话线传输的模拟信号的过程; 解调则是一个反调制的过程, 它把通过电话线传输来的已调制的模拟信号转变为计算机能够识别的二进制数字信号。正是通过这样一个调制与解调的数模转换过程, 使两台计算机之间可以进行远程通信。数字信号调制/解调的工作, 即数字信号到模拟信号的转换完全由 Modem 上的 DSP (数字信号处理) 芯片执行。计算机中的 CPU 只负责把数字信号传递给 Modem 上的 DSP 芯片, 不参与其中的工作。

相对于波形的幅值、频率和相位 3 大特性, Modem 有 3 种调制方法。

□ 幅移键控 (ASK): 调幅调制, 以正弦波的幅度在两种幅值之间变换表示数字信号 1 和 0。它可用 16 个不同的相位和幅度电平, 可以使 1200bit/s 的 Modem 传送 19200bit/s 的数

据信号。该种 Modem 一般用于高速同步通信中。

□ 频移键控 (FSK): 调频调制, 以正弦波的幅度在两种频率之间变换表示数字信号 1 和 0。用特殊的音频范围来区别发送数据和接收数据。如调频 Modem Bell-103 型发送和接收数据的二进制逻辑被指定的专用频率是: 发送时, 信号逻辑 0、频率 1070Hz, 信号逻辑 1、频率 1270Hz; 接收时, 信号逻辑 0、频率 2025Hz, 信号逻辑 1、频率 2225Hz。

□ 相移键控 (PSK): 调相调制, 以正弦波的幅度在两种相位之间变换表示数字信号 1 和 0。高速的 Modem 常用四相制、八相制, 而四相制是用四个不同的相位表示 00、01、10、11 四个二进制数。如调相 Modem Bell-212A 型。该技术可以使 300bit/s 的 Modem 传送 600bit/s 的信息, 因此在不提高线路调制速率仅提高信号传输速率时很有意义, 但控制复杂, 成本较高, 八相制更复杂。

1.3.2 Modem 的传输速率

Modem 的传输速率, 指的是 Modem 每秒钟传送的数据量大小。我们平常说的 14.4kbit/s、28.8kbit/s、33.6kbit/s 等, 指的就是 Modem 的传输速率。传输速率以 bit/s (比特/秒) 为单位。因此, 一台 33.6kbit/s 的 Modem 每秒钟可以传输 33600bit 的数据。由于目前的 Modem 在传输时都对数据进行了压缩, 因此 33.6kbit/s 的 Modem 的数据吞吐量理论上可以达到 115200bit/s, 甚至 230400bit/s。

Modem 的速率有两个: 一个是 DCE 速率 (线上速率), 指两个 Modem 连线时两者间的通信速率。通常所说的 14.4kbit/s、28.8kbit/s、33.6kbit/s 等几种就指的是线上速率, 另一个是 DTE 速率 (终端速率), 是指 Modem 与计算机之间的通信速率。我们讲的 19.2kbit/s、38.4kbit/s、57.6kbit/s 指的是终端速率。Modem 的速度当然是越快越好, 但是速度快的价格相对高些。

Modem 实际的传输速率主要取决于以下几个因素:

1. 电话线路的质量

因为调制后的信号是经由电话线进行传送, 如果电话线路质量不佳, Modem 将会降低速率以保证准确率。在连接 Modem 时, 要尽量减少连线长度, 多余的连线要剪去, 切勿绕成一圈堆放。另外, 最好不要使用分机, 连线也应避免在电视机等干扰源上经过。

2. 是否有足够的带宽

如果在同一时间上网的人数很多, 就会造成线路的拥挤和阻塞, Modem 的传输速率自然也会随之下降。因此, ISP 是否能提供足够的带宽非常关键。另外, 避免在繁忙时段上网也是一个解决方法。尤其是在下载文件时, 在繁忙时段与非繁忙时段下载所费的时间会相差很多。

3. 对方的 Modem 速率

Modem 所支持的调制协议是向下兼容的, 实际的连接速率取决于速率较低的一方。因此, 如果对方的 Modem 是 14.4kbit/s 的, 即使你用的是 56kbit/s 的 Modem, 也只能以 14.4kbit/s 的速率进行数据传输。

1.3.3 Modem 的类型

Modem 从外观上分为两类，一类是内置 Modem，另一类是外置 Modem。内置的 Modem 与外置的 Modem 的最大区别是一个插在计算机机箱的插槽里，一个放在计算机外面。内置的 Modem 与显示卡和声卡一样是一块电路板，不用额外的电源，它除了与电话线连接外不需要任何的连线。后面会介绍 Modem 的安装。

外置的 Modem 有自己独立的外壳，要通过串行电缆与计算机的串行接口连接，要通过变压器来把 220V 的电压变为 9V 或 12V。通过外置 Modem 的指示灯可以了解 Modem 的工作状态，比如是否工作正常、是否正在传输数据等。

调制解调器按工作方式分类：

1. 频带调制解调器

频带调制解调器是利用数字基带信号对模拟信号进行调制，信道工作频率在音频信号范围内（300Hz~3400Hz），其传输速率较低，一般在 300bit/s 到 28.8kbit/s 之间。由于频带调制解调器可工作在音频信号的频率范围内，因此可以用于程控交换机。故频带调制解调器有两种工作方式，拨号方式与专线方式。

2. 基带调制解调器

基带调制解调器对于数据信号进行信道编码，使数据信号变为适应信道传送的基带信号，其传输信道一般为实线，且传输速率较高，一般从 64kbit/s 到 2Mbit/s。基带调制解调器只能工作于专线方式下。

由于基带调制解调器使用的技术为各厂家独自开发，因此基带调制解调器无协议标准，因而基带调制解调器必须由厂家成对提供，不同厂家的基带调制解调器之间一般不能通信。

与基带调制解调器不同，频带调制解调器有国际协议标准，因而具有相同协议的调制解调器之间可以互通。下面一节详细讲解 Modem 通信协议体系的类型。

1.3.4 Modem 的通信协议体系

ITU 即以前的 CCITT（法语 Comit Consultatif International de Telegraphiqueet Telephonique，国际电报电话咨询委员会）制定了一些通信方面的标准。这些协议是推荐给通信业的制造商和软件开发商采用的，其中所有关于小型（微型）计算机的推荐标准都以 V 或 X 打头，其中 V 系列用于电话交换网，X 系列用于非电话交换网系统；修订或变更的协议则加上 bis（第二次）或 ter（第三次）后缀。另外，一些大公司自己推出的一些标准由于市场占有率高也成为了事实上的标准。上述两类标准构成了常见的 Modem 通信协议体系，越来越多的公司采纳这些推荐标准。

常见的 Modem 通信协议体系可以分为调制协议（Modulation Protocols）、差错控制协议（Error Control Protocols）、数据压缩协议（Data Compression Protocols）、文件传输协议、传真协议和语音协议等几大类。

1. 调制协议

平时在 Modem 的包装盒或说明书上看到的 V.32、V.32bis、V.34、V.34+等等，指的就是

Modem 所采用的调制协议。

Bell 103 贝尔的 300bit/s 标准, 相当于 V.21。由于与 ITU 标准不兼容, 已被淘汰。

Bell 212 贝尔的 1200bit/s 标准, 相当于 V.22。由于与 ITU 标准不兼容, 已被淘汰。

V.21 ITU 调制协议, 最高速率 300bit/s。已被淘汰。

V.22 ITU 调制协议, 最高速率 1200bit/s。已被淘汰。

V.22bis ITU 调制协议, 最高速率 2400bit/s。已被淘汰。

V.27DPSK ITU 调制协议, 最高速率 4800bit/s。支持同步专用模式。

V.29QAM ITU 调制协议, 最高速率 9600bit/s。支持同步专用模式。

V.32 ITU 调制协议, 最高速率 9600bit/s。采用 32-TCM 调制技术

V.32bis ITU 调制协议, 最高速率 14400bit/s。采用 128-TCM 多复合调制技术, 支持同步、异步、专线及全双工两线拨号方式。为固定调制模式, 能自适应线路质量。

V.FC (V.Fast Class) 调制协议, 最高速率 19200bit/s。采用的也是 V.34 回波抵消技术及多重复合技术。它是一个非标准协议, 但未被 ITU 正式批准, 已被淘汰。

V.34 ITU 调制协议, 最高速率 28800bit/s。采用回波抵消及多重复合技术, 支持全双工两线模式, 具有智能型自适应功能, 能自适应线路质量。

V.34bis 调制协议, 最高速率 33600bit/s。1996 年才出现的一种新标准。

V.90 ITU 调制协议, 最高速率 56000bit/s。

56K Modem 的调制协议标准已提出多年, 但由于长期以来一直存在以 Rockwell 为首的 K56flex 和以 U.S.Robotics 为首的 X2 的两种互不兼容的标准, 使得 56K Modem 迟迟得不到普及。在国际电信联盟的努力下, 56K Modem 的标准终于统一为 ITU V.90。V.90 是传统意义上调制解调器协议的终结。电话网在交换机和终端用户之间的频宽只有 4kHz。根据采样定理, 采样率是 4kHz/s 的两倍——8kHz/s。每一次采样得到 8bit 数据, 其中一个 bit 是交换机同步数据位, 可以理解为符号位, 它是无效数据, 所以每一次采样只能拿到 7bit 的数据, 共 56kbit。这就是 56K Modem 的来历。而 4kHz 的频宽是 PSTN (Public Switched Telephone Network 公用电话交换网) 的极限, 所以传统 Modem 理论上的最高速度是 56kbit/s。这是 V.90 协议 Modem 下载数据的情况。在 Modem 上传数据时, 数据量会被 Modem 进行一次 D/A 转换, 到达交换机后, 再做一次 A/D 转换。因为有两处转换的误差, 所以 V.90 Modem 的上传速度极限是 33.6kbit/s。

2. 差错控制协议

随着 Modem 传输速率的不断提高, 电话线路上的噪声、电流的异常突变等, 造成数据传输出错的现象越来越严重。差错控制协议要解决的就是如何在高速传输中保证数据的准确率。目前的差错控制协议有两个工业标准: MNP4 和 V4.2。其中 MNP (Microcom Network Protocols) 是 Microcom 公司制定的传输协议, 包括了 MNP1~MNP10。由于商业原因, Microcom 目前只公布了 MNP1~MNP5, 其中 MNP4 是目前被广泛使用的差错控制协议之一。而 V4.2 则是国际电信联盟制定的 MNP4 改良版, 它包含了 MNP4 和 LAP-M 两种控制算法。因此, 一个使用 V4.2 协议的 Modem 可以和一个只支持 MNP4 协议的 Modem 建立无差错控制连接, 而反之则不能。所以我们在购买 Modem 时, 最好选择支持 V4.2 协议的 Modem。

3. 压缩协议

数据压缩 (Data Compression) 实际上是对传输文件进行“压缩”，使之变成最短代码再进行传送，以求增加吞吐量 (Throughput)。压缩比实际上是根据对一些有代表性的文件进行传输处理时按照加权方式计算出来的，它被作为一个参考值。

为了提高数据的传输量，缩短传输时间，现在大多数 Modem 在传输时都会先对数据进行压缩。与差错控制协议相似，数据压缩协议也有两个工业标准：MNP5 和 V4.2bis。MNP5 采用了 Run-Length 编码和 Huffman 编码两种压缩算法，最大压缩比为 2:1。V4.2bis 采用了 Lempel-Ziv 压缩技术，最大压缩比可达 4:1。这就是 V4.2bis 比 MNP5 要快的原因。要注意的是，数据压缩协议是建立在差错控制协议的基础上，MNP5 需要 MNP4 的支持，V4.2bis 也需要 V4.2 的支持。不过，虽然 V4.2 包含了 MNP4，但 V4.2bis 不包含 MNP5。

4. 文件传输协议

对普通 Modem 用户而言，Modem 最大的一个用途就是文件传输。在双方进行文件传输之前，首先必须协商相互间的通信规程，即文件传输协议 (File Transfer Protocol, 简称为 FTP)。文件传输协议是一个规则集，用来描述和控制计算机之间相互传送文件的过程。这个协议包括了文件的识别、传送的起止时间、错误的判断与纠正等内容。

串行通信的文件传输协议有许多种，主要有 ASCII、XModem、YModem、ZModem 和 Kermit 等通信协议，下面分别进行介绍。

(1) ASCII 通信协议

ASCII 通信协议是最快的传输协议，但只能传送文本文件。

(2) XModem 协议

XModem 协议是由 Ward Christensen 在 70 年代提出并实现的，是最早出现的一种用于 PC 机的文件传输协议，至今仍被广泛应用。

XModem 传输的数据单位是信息包，包含一个标题开始字符 SOH，一个单字节包序号，一个包序号的补码，128 个字节数据和一个单字节的校验和。它把数据划分成 128 个字符的小“包”进行发送，每发送一个小“包”，都要检验是否正确。如果发现有错，则再重发该小“包”，否则继续发送下一个小“包”，直至整个文件传输完毕。因此 XModem 是一种发送等价协议，具有流量控制功能。

XModem 的主要优点是简单、通用，大部分通信软件都支持该协议。

缺点：XModem 传输的每个信息包只有 128 个字节，采用出错重传方式，因而文件传输很慢。它对于 300bit/s 和 1200bit/s 的慢速 Modem 很适合。另外，XModem 在一次传输中只能发送或接收一个文件。

XModem 协议非常适合在电话线路质量较差的情况下使用。

(3) YModem 协议

80 年代初出现的 YModem 协议是由 XModem 协议演变而来的，较 XModem 协议在效率和可靠性等方面均有很大改进。

YModem 信息包中的数据段长度最大为 1024 个字节，并且在一次传输中可以发送 1024 个字节和 128 个字节混合的块。由于信息包的大小是 XModem 的 8 倍，所以传输速度有较大的提高。当数据块被正确接收时不作应答，只有当数据块在传送时出错时才发出 NAK 应答，

通知此数据块需要重新发送。当传输线路性能不好而使误码增加时, YModem 协议能自动将数据段长度降为 128 个字节。

YModem 和 XModem 协议类似, 但是 YModem 提供了批处理模式, 即在一次传输中可以发送或接收几个文件。但是 YModem 协议是在 XModem 的基础上改进而来的, 因而 XModem 的一些固有缺点在 YModem 协议中依然存在。例如在只支持 7 位数据的系统上无法传输 8 位数据。

(4) ZModem 协议

ZModem 协议并不是由 XModem 或 YModem 协议演变而来, 它不但具有纠错功能, 而且还是一种流式协议, 常用于 BBS (电子公告牌)。

ZModem 协议不再以“包”来分割发送数据, 而是按连续的数据流进行处理, 使错误检查码遍布文件的自始至终。接收方对整个文件做检验, 如果有错, 它只将出错部分重发。因此 ZModem 协议比 XModem 和 YModem 两种协议效率高得多, 速度也有明显提高。

ZModem 协议能够在 7 位的通路上用编码的控制字符和其他特殊字符传输 8 位数据。同时它也支持一次传输多个文件, 能用通配符“.”和“?”号。

另外符合 ZModem 协议的通信软件能够自动识别出一个 ZModem 文件传输的开始序列, 从而自动开始接收。如果文件在传输过程中因故障而中断, ZModem 会记住文件的中断点, 重发时, 它会从中断点继续发送, 而不用重发整个文件。

(5) Kermit 协议

Kermit 协议是哥伦比亚大学开发的, 主要用于微型计算机和大型机之间文件传输。Kermit 协议在多种电话线通信软件包中使用。协议规定:

信息以可变长度包 (通常其长度可达 96 字节) 的形式进行传输, 并检查每个包的传输错误。控制字符被转换为标准的可打印 ASCII 字符, 因而可以进行传送, 同时避免了在接收端引起误解的危险。

Kermit 协议也是一种发送并等待的包协议, 因而它类似于 XModem, 但它对 XModem 进行了一些改进。例如在 7 位通路上传输 8 位数据, 并且 Kermit 协议可以一次传输多个文件。Kermit 协议还提供了压缩功能, 采用的错误检测方式也比 XModem 协议先进。

Kermit 协议的最大的优点是对通信环境的要求非常低, 适应性非常强, 几乎在所有的系统上都能实现。但是其最大缺点是效率低, 文件传输的速度慢。因而一般将 Kermit 作为最后的手段, 在没有其他可适用的协议时才使用它。

5. 传真协议

V.17 ITU 传真协议: 较新的标准, 最高速率为 14400bit/s。

V.29 ITU 传真协议: 它是 Group3 的规范化, 最高速率为 9600bit/s。Group 最高速率为 4800bit/s。

6. 其他协议

V.8 ITU 握手协议: 它是专为 V.34 制定的规程。

V.25 ITU 数据分组通信协议: 应用在 Telnet 和 Tymnet 系统上。

V.25bis ITU 拨号协议: 支持同步和 HDLC 传输格式。

1.3.5 Modem 的安装与使用

下面介绍 Modem 的硬件安装。

1. 外置式 Modem 的安装

(1) 连接电话线: 把电话线的 RJ11 插头插入 Modem 的 Line 接口, 再用电话线把 Modem 的 Phone 接口与电话机连接。

(2) 关闭计算机电源, 将 Modem 所配电缆的一端 (25 针阳头端) 与 Modem 连接, 另一端 (9 针或者 25 针插头) 与主机上的 COM 口连接。

(3) 将电源变压器与 Modem 的 POWER 或 AC 接口连接。接通电源后, Modem 的 MR 指示灯应长亮。如果 MR 灯不亮或不停闪烁, 则表示未正确安装或 Modem 自身故障。对于带语音功能的 Modem, 还应把 Modem 的 SPK 接口与声卡上的 Line In 接口连接, 当然也可直接与耳机等输出设备连接。

另外, Modem 的 MIC 接口用于连接驻极体麦克风, 但最好还是把麦克风接到声卡上。

2. 内置式 Modem 的安装

(1) 根据说明书的指示, 设置好有关的跳线。由于 COM1 与 COM3、COM2 与 COM4 共用一个中断, 因此通常可设置为 COM3/IRQ4 或 COM4/IRQ3。

(2) 关闭计算机电源并打开机箱, 将 Modem 卡插入主板上一个空置的扩展槽。

(3) 连接电话线: 把电话线的 RJ11 插头插入 Modem 卡上的 Line 接口, 再用电话线把 Modem 卡上的 Phone 接口与电话机连接。此时拿起电话机, 应能正常拨打电话。

下面介绍 Modem 的驱动程序安装。

当硬件安装完成后, 打开计算机, 外置式 Modem 还应打开 Modem 的开关。对于大多数 Modem, Windows 98 会报告“找到新的硬件设备”, 此时只需选择“硬件厂商提供驱动程序”, 并插入 Modem 的安装盘即可。如果 Windows 98 启动后未能侦测到 Modem, 也可以按以下步骤完成安装。

(1) 进入 Windows 98 的“控制面板”窗口, 双击“调制解调器”图标; 在“调制解调器选项”窗口中单击“调制解调器”项, 单击“添加”按钮;

(2) 选中“不检测调制解调器, 而将从清单中选定一个”选项, 然后单击“下一步”按钮;

(3) 在 Modem 列表中选择相应的厂商与型号, 然后单击“下一步”按钮; 或者插入 Modem 的安装盘后, 单击“从磁盘安装”按钮进行安装也可以。要证明 Modem 是否安装成功, 可使用 Windows 98 附件中的电话拨号程序随便拨打一个电话, 如果成功的话, 说明 Modem 已被正确安装。对于上网用户, 还需要安装拨号网络和协议。

1.3.6 外置式调制解调器的指示灯

在正确安装 Modem 之后, Modem 即可加电。当电源打开后, Modem 前面板 HS 和 MR 灯亮, 各指示灯状态及含义如下所示。

通用的外置式调制解调器指示灯含义的描述见表 1.1。

表 1.1

外置式调制解调器指示灯含义

指示灯	描述
HS (高速)	当 Modem 在其最高速率操作时, HS 灯亮
AA (自动应答)	当 Modem 被设为应答方式时, AA 灯亮。当 Modem 检测到呼叫, 而它被置于自动应答方式时, 在振铃期间, AA 灯灭。如不是自动应答方式, 在振铃期间, AA 灯亮
CD (载波检测)	当本地 Modem 接收到远地 Modem 的载波信号或 DCD 被设为 HIGH 时, 此灯会亮
OH (摘机)	当 Modem 摘机与拨号线相连时, 此灯会亮
SD (发送数据)	当 Modem 发送数据时, 此灯将会闪烁
RD (接收数据)	当 Modem 接收数据时, 此灯将会闪烁
TR (终端准备)	当检测到终端发出的 DTR 信号时, 此灯会亮
MR (Modem 就绪)	当 Modem 上电时亮。当 Modem 在自检或诊断方式时, 此灯将会闪烁

1.3.7 Modem 技术的新发展

1. 具有多种功能的 Modem

目前, Modem 的生产厂家为适应计算机和信息产业迅速发展的需要, 已经开发出了有 VOICE (语音)、DSVD (同步语音)、FAX (传真) 功能的 Modem。具有 VOICE 功能的 Modem 可以把计算机当作“数字录音电话”来使用, 具有 DSVD 功能的 Modem 可以进行电话交谈, 具有 FAX 功能的 Modem 可以把计算机当作“传真机”来使用。现在还有速率在 64kbit/s 到 128kbit/s 的 ISDN (综合业务数字网) 调制解调器, 速率达 30Mbit/s 线缆调制解调器, 称为 Cable Modem。另外还有一种速率在 640kbit/s 到 8Mbit/s 之间的 ADSL (非对称数字用户线调制解调器) 调制解调器。

Cable Modem 是在有线电视电缆上对数据进行调制, 然后在有线网的某个频率范围内进行传输, 接收一方再在同一频率范围内对该已调制的信号进行解调, 解析出数据, 传递给接收方。它采用一种视频信号格式来传送 Internet 信息。它不但具有调制解调功能, 还集路由器、集线器、桥接器于一身, 理论传输速度更可达 10Mbit/s 以上。通过 Cable Modem 上网, 无须拨号上网, 不占用电话线, 可永久连接。每个用户都有独立的 IP 地址, 相当于拥有了一条个人专线。Cable Modem 技术现在已经成熟, ITU 也批准了 MCNS (多媒体电缆网络系统) 标准。

ADSL 调制解调技术利用现有电话铜线基础设施几乎就能为所有家庭和企业提供各种电信服务, 允许用户以比现今最新的 56K Modem 高 100 倍左右的速率通过数据网络或 Internet 以及相关服务进行交互式通信。在这种交互式通信中, ADSL 的下行线路可提供比上行线路更高的带宽, 即上下行带宽不相等, 且一般都在 1:10 左右。如果线路的上行速率是 640kbit/s, 则下行线路就有 6.4Mbit/s 的高速传输速率。这也就是 ADSL 为什么叫非对称数字用户线的原因。同时, 由于 ADSL 采用频分复用技术, 可将电话语音和数据流一起传输, 用户只需加装一个 ADSL 用户端设备, 通过分流器 (语音与数据分离器) 与电话并联, 便可使一条普通电话线同时通话和上网, 而且互不干扰。

2. 软件 Modem

目前出现了一种支持 Rockwell 56kbit/s Flex 芯片组的 56kbit/s 的软件 Modem。

软件 Modem 是以软件为基础, Modem 里面还是有芯片, 只是把传统的 DSP 技术变为更先进的主机信号处理 (HSP) 技术, 随着微处理器功能的越来越强大, 特别是自 Pentium MMX 芯片的推出以后, 一些以往 CPU 所不能处理的工作现在很容易就可以解决了。CPU 担负了普通 Modem 中 DSP 芯片的一部分工作, 把数字数据流进行处理后送给数/模转换芯片进行调制。软件 Modem 需要驱动程序的支持, 当计算机的操作系统改变或速度升级, 驱动程序就需要随着改变或升级。

硬件 Modem 与软件 Modem 两者各有优势。软件 Modem 芯片组的成本只有硬件 Modem 的一半左右, 可以节省不少资金, 因此更容易被市场接受。但是随着硬件 Modem 芯片组价格的降低, 二者的价格也正在逐步接近。

本章小结

本章首先介绍了数据通信系统的组成、同步通信、异步通信、波特率、DTE 设备和 DCE 设备等一些基本概念, 然后介绍了重要的 DCE 设备——Modem 的基本原理、传输速率、分类、通信协议体系、安装和使用、技术的新发展, 其中重点介绍了 Modem 的通信协议体系。掌握这些内容有助于从总体上了解异步通信应用的结构和采用的技术。

从下一章开始将着重介绍 Modem 的 Hayes 标准以及 PC 机的异步串行接口等技术。

第 2 章 通用串行通信标准和通用 Modem 命令

本章主要内容:

- RS-232C 标准
- AT 命令
- 通用异步接收发送器 UART

RS-232C (RS 本是 Recommend Standard 的缩写) 是一种历史悠久的计算机接口标准, 它于 1969 年被国际组织认可。

本章介绍了串行通信标准中的 RS-232C 标准和在 PC 机 Modem 领域被广为支持的 AT 命令集, 最后简单介绍了通用异步接收发送器 UART。

2.1 RS-232C 标准

所谓串行通信接口标准, 是指串行通信接口与外设的信号连接标准。

实际中常用的串行通信接口标准有 3 种: RS-232C, RS-422A/423A 和 20mA 电流环。常用的 PC 机都配置了 RS-232C 标准接口。RS-232C 标准常简称为 RS-232。

下面介绍一下 RS-232C 标准接口。RS-232C 的定义包括电气特性 (如电压值)、机械特性 (如接头形状) 及功能特性 (如脚位信号) 等。它允许一个发送设备连接到一个接收设备以传送数据, 其原始规范的最大传输速度为 20kbit/s, 但事实上, 现在的应用早已超过这个速度范围。RS-232C 可说是相当简单的一种通信标准, 若不使用硬件流量控制, 则只需利用 3 根信号线, 便可做到全双工的传输作业。RS-232C 的电气特性属于非平衡传输方式, 抗干扰能力较弱, 故传输距离较短, 约为 15m 左右。

串行通信接口基本功能是: 在发送时, 把 CPU 送来的并行码转换成串行码, 逐位地依次发送出去; 在接收时, 把发送过来的串行码逐位地接收, 组装成并行码, 并行地发送给 CPU 去处理。这种串行到并行转换的功能, 当然可以用软件来实现, 但是这样会降低 CPU 的利用率, 所以常用硬件电路来实现这种功能, 这种硬件电路叫做串行通信接口。

普通的 Modem 通常都是通过 RS-232C 串行口信号线与计算机连接。RS-232C 标准说明的是 DTE 与 DCE 之间的连接规定, 包括两设备接口电路的机械特性、信号线功能描述以及电信号特性。

根据 RS-232C 标准规定, 接口电路采用一对物理 D 型连接器: DTE 设备应该有一个 D 型插头接口, DCE 设备应该有一个 D 型插座接口。D 型连接器可以是 25 芯 (简称为 DB25),

也可以是 9 芯（简称为 DB9）。RS-232C 引脚分配如图 2.1 所示。

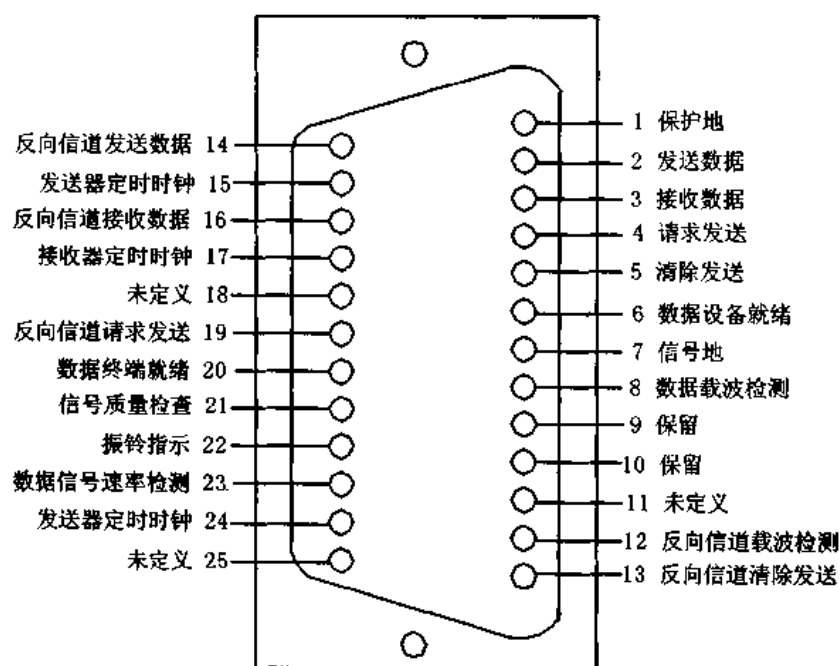


图 2.1 RS-232C 引脚分配

2.1.1 信号连接

RS-232C 规定使用一种 DB25 连接器，其中 20 个脚作了定义，9、10、11、18、25 未作定义。各引脚上的信号规定详见表 2.1。表中的反向信道在一般串行通信中很少用到。

表 2.1

RS-232C 连接器引脚信号定义

引脚	信号名称	信号方向	简称	信号功能
1	保护地		PG	作为设备的保护接地
2	发送数据	DTE→DCE	TxD	DTE 向 DCE 发送串行数据
3	接收数据	DCE→DTE	RxD	DTE 从 DCE 接收串行数据
4	请求发送	DTE→DCE	RTS	打开 DCE 的发送器
5	清除发送	DCE→DTE	CTS	响应 DTE 请求，提示 DCE 开始发送
6	数据设备准备就绪	DCE→DTE	DSR	提示 DCE 已接上信道，不处于测试、对话或拨号模式
7	信号地		SG	整个电路的公共信号地
8	数据载波检测	DCE→DTE	DCD	DCE 接收到远程载波，通信链路已连接
12	反向信道载波检测	DCE→DTE		DCE 接收到远程载波，通信链路已连接
13	反向信道清除发送	DCE→DTE		响应 DTE 请求，提示 DCE 开始发送
14	反向信道发送数据	DTE→DCE		发送低速率数据

续表

引脚	信号名称	信号方向	简称	信号功能
15	发送器定时时钟	DCE→DTE		给 DTE 提供发送时钟
16	反向信道接收数据	DCE→DTE		接收低速率数据
17	接收器定时时钟	DCE→DTE		给 DTE 提供接收时钟
19	反向信道请求发送	DTE→DCE		打开 DCE 的发送器
20	数据终端就绪	DTE→DCE	DTR	表明 DTE 已准备就绪
21	信号质量检查	DCE→DTE		指示接收的误码率合格
22	振铃指示	DCE→DTE	RI	线路上有振铃
23	数据信号速率检测	DTE→DCE	DSRD	选择较高的速率, 双向通知
24	发送器定时时钟			给 DCE 提供发送时钟

RS-232C 串行口信号分为 3 类: 传送信号、联络信号和信号地。

(1) 传送信号 (TxD 和 RxD)

传送信号是经由 TxD (发送数据信号线, 引脚 2) 传送和 RxD (接收数据信号线, 引脚 3) 接收的信息格式即一个传送单位 (字节) 由起始位、数据位、奇偶校验位和停止位组成。

(2) 联络信号 (RTS、CTS、DTR、DSR、DCD 和 RI 等 6 个信号)

6 个信号各自功能为:

RTS (请求传送, 引脚 4), 是 PC 向 Modem 发出的联络信号。高电平表示 PC 机请求向 Modem 传送数据。

CTS (清除发送, 引脚 5), 是 Modem 向 PC 机发出的联络信号。高电平表示 Modem 响应 PC 发出的 RTS 信号, 且准备向远端 Modem 发送数据。

DTR (数据终端就绪, 引脚 20), 是 PC 向 Modem 发出的联络信号。高电平表示 PC 机处于就绪状态, 本地 Modem 和远端 Modem 之间可以建立通信信道。若为低电平, 则强迫 Modem 终止通信。

DSR (数据装置就绪, 引脚 6), 是 Modem 向 PC 机发出的联络信号。它指出本地 Modem 的工作状态, 高电平表示 Modem 没有处于测试通话状态, 可以和远端 Modem 建立通道。

DCD (传送检测, 引脚 8), 是 Modem 向 PC 发出的状态信号。高电平表示本地 DCE 接收到远端 Modem 发来的载波信号。

RI (铃指示, 引脚 22), Modem 向 PC 发出的状态信号。高电平表示本地 Modem 收到远端 Modem 发来的振铃信号。

(3) SG (信号地)

SG (信号地, 引脚 7) 为相连的 PC 和 Modem 提供同一电势参考点。

2.1.2 握手

由上可知 DTE 和 DCE 之间如果实现双向通信, 至少需要 3 条信号线: TxD 使数据从 DTE 到 DCE, RxD 使数据从 DCE 到 DTE, SG 为信号地。

但在实际中的许多情况下，发送设备一方需要知道接收设备一方是否已经做好接收准备。最典型的例子就是打印机与计算机之间的通信。由于打印机的打印速度远远低于计算机的发送数据速度，因此为了能正确接收数据，打印机必须能够通知计算机暂停发送，采用的方法就是使用握手信号。握手信号还常用于打印机没有打印纸、一台计算机向另一台计算机发送数据而接收速度赶不上发送速度等情况。因此，必须使用握手信号，它提供了一种控制数据流的方法，即接收设备可以控制发送设备的数据发送。同时也说明如果接收设备速度比发送速度快，则握手信号可以略去。

在异步串行通信中，这称之为握手（handshaking）或流量控制（flow control）。握手控制可以具体分为硬件握手（硬件流控）和软件握手（软件流控）。

1. 硬件握手

硬件握手是使用专门的握手电路去控制数据的传输。当接收设备准备好之后，就通过专用的握手电路传送一个正电压给发送设备，指示发送设备发送数据。如果接收设备传送一个负电压给发送设备，则指示发送设备停止发送数据。

因此为了构成硬件握手，还必须增加一条传送握手信号的电路。这样，为了完成数据通信需要有 3 类电路：数据线、信号线和握手线。

（1）DTE 到 DCE

为了控制 DTE 的发送数据，DCE 使用 DSR 信号作为主握手信号去通知 DTE 已做好接收数据的准备。当通知 DTE 暂停发送数据时，置 DSR 无效。

DCE 还使用 CTS 信号作为第二握手信号（辅助握手信号）控制 DTE 设备。仅当这两条握手线都有效（高电平）时，DTE 才发送数据。

（2）DCE 到 DTE

为了控制 DCE 的数据发送，DTE 使用 DTR 信号作为主握手信号去通知 DCE 已做好接收数据的准备。当通知 DCE 暂停发送数据时，置 DTR 无效。

DTE 还使用 RTS 信号作为第二握手信号控制 DCE 设备。仅当这两条握手线都有效时，DCE 才发送数据。

（3）双向通信

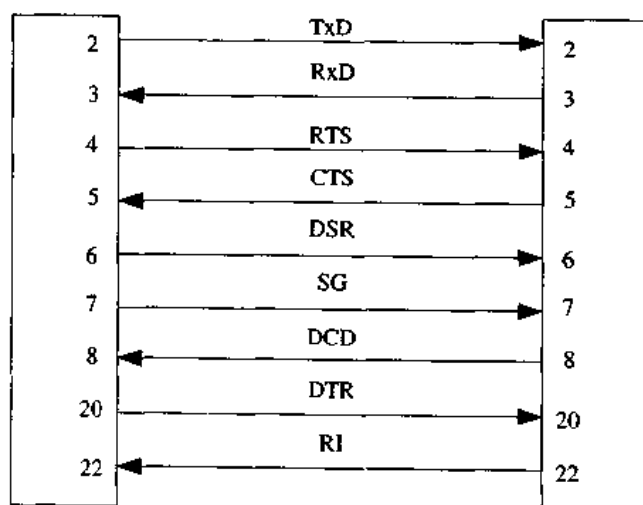


图 2.2 有握手功能的双向通信

双向通信中只使用主握手线, 则共需要 5 条信号线: TxD、RxD、DSR、DTR 和 SG。如果还使用第二握手线, 则共需要 7 条信号线。

为了使 DCE 能向 DTE 提供更多的信息, 通常还使用 RI 和 DCD 两条信号线。这样一个完整的异步串行通信所必需的就是这 9 条信号线, 如图 2.2 所示。

这 9 条信号线是最常用的, 这就是个人计算机串口通常采用 9 针而不是 25 针接口的原因。25 针的接口是 RS-232C 功能所必需的。

2. 软件握手

软件握手的原理机制与硬件握手基本相同, 不同的是握手信号是在数据线 (TxD 和 RxD) 上进行传送的, 而不是在专门握手线上传送。这是因为软件握手信号是由特殊字符组成的, 所以传送这些字符必须使用数据电路, 而不是使用专门握手电路。这种方法常用在直接连接或通过 Modem 连接的两台计算机之间进行双向通信的场合。

软件握手最常用的协议是 XON/XOFF 协议。该协议主要解决通信双方处理速度不匹配的问题, 协议规定发送 XOFF 表示暂停发送数据, 发送 XON 表示继续发送数据。

如果要使发送设备停止发送数据, 接收设备只需发送一个 ASCII 字符 DC3 (十进制 19, 十六进制 13) 给发送设备, 该字符称为 XOFF 字符。

如果要发送设备继续发送数据, 则接收设备只需发送一个 ASCII 字符 DC1 (十进制 17, 十六进制 11) 给发送设备, 该字符称为 XON 字符。

在实际应用中, 常使用一个数据接收缓冲器。当接收缓冲器将上溢 (装满) 时, 接收设备发送一个 XOFF 给发送设备; 当接收缓冲器将下溢 (快空) 时, 则向发送设备发送一个 XON 字符。

3. 硬件与软件相结合的握手

软件握手有个明显的缺点: DCE 设备不能传递 XON 和 XOFF 字符。这就是说, 当用户试图传递含有 XOFF 字符的数据流时, DCE 将出现“冻结”, 即 DCE 将停止向 DTE 发送数据。因而在一般情况下 DTE 与 DCE 之间不采用软件握手。

为了综合硬件握手和软件握手的好处, 可以采用硬件和软件相结合的握手控制。假设 DTE 设备为计算机, DCE 设备为 Modem, 两台计算机之间通过 Modem 经电话线连接, 则此时计算机与 Modem 之间可采用硬件握手方法, 而两台计算机之间可以使用软件握手方法进行联系。

2.1.3 微机的 RS-232C 接口

个人计算机的 RS-232C 接口名称有多个: RS-232C 口、串口、通信口、COM 口、异步口等。

目前 DOS3.3 以上版本和 Windows 3.2/98/NT 最多支持 4 个串口: COM1、COM2、COM3 和 COM4。

它们所占用的 I/O 口地址和中断号见表 2.2。

表 2.2 PC 机的 4 个串口

串口	I/O 地址	中断号
COM1	0x3f8	IRQ4
COM2	0x2f8	IRQ3
COM3	0x3e8	IRQ4
COM4	0x2e8	IRQ3

为了更好地说明 RS-232C 接口电路的实际工作情况,下面以应答呼叫过程为例,具体分析其信号间的交互关系。

所谓应答呼叫过程,即指 Modem 从接收到振铃信号开始,到数据传输结束后 Modem 和 DTE 恢复到原来的空闲状态为止的过程。假设 Modem 是全双工工作,因此不需要 RTS/CTS 握手信号,即 DTE 总是保持 RTS ON 状态,而 Modem 总是保持 CTS ON 状态。其他控制信号初始化时均处于 OFF 状态。ON 指该信号有效,OFF 指该信号无效。

(1) 数据终端 DTE 的控制软件持续监视振铃指示 (RI),等待该信号有效。DTE 和 Modem 的引脚连线如图 2.3 所示。

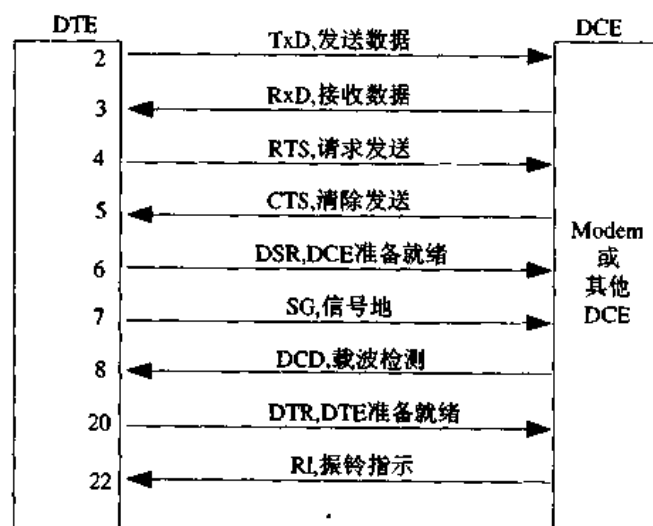


图 2.3 DTE 和 Modem 的引脚连线

(2) 响铃后, Modem 在振铃脉冲期间发出振铃指示信号 (RI 有效), 在振铃脉冲间隔期间, 振铃指示信号无效。即随着振铃脉冲的有无, RI 信号 ON/OFF 交替变化。

(3) DTE 的通信控制软件在检测到振铃指示后, 开始通过计算振铃指示 ON/OFF 变化的次数对振铃进行计数。当达到程序预置好的振铃数时, 控制软件发出数据终端就绪信号 (DTR 有效), 迫使 Modem 进入摘机状态, 开始应答电话。

(4) Modem 在等待一小段时间 (如 2 秒钟) 后, 自动地发送它的应答载波信号。同时 Modem 发出数据设备就绪信号 (DSR 信号有效), 通知 DTE 已完成所有准备工作, 正在等待对方载波信号。

(5) 在 DTE 发出数据终端就绪信号 (DTR 有效) 期间, DTE 的控制软件监视数据设备就绪信号 (DSR 是否有效)。当 DSR 变为 ON 状态后, DTE 便知道了 Modem 已准备建立数据链路, 于是 DTE 开始监视载波检测 (DCD) 信号, 以检查数据链路是否已建立。

(6) 当主叫 Modem 的载波信号出现在电话线上时, 被叫 Modem 就发出载波检测信号 (DCD), 通知 DTE 已建立数据链路。

(7) 在数据链路连接期间, 发送数据 (TxD) 和接收数据 (RxD) 线上即开始了全双工通信。同时, DTE 仍监视着载波检测 (DCD) 信号, 以确定数据链路是否连接。

(8) 数据传输结束后, DTE 使数据终端就绪信号 (DTE 无效), Modem 撤消载波信号并以载波检测 (DCD) 和数据设备就绪 (DSR) 信号无效给予响应。数据链路释放后, Modem 和 DTE 准备下一次接收或作另一次呼叫。

2.2 通用 Modem 命令

各厂家生产的 Modem 除了可以执行最基本的 AT 命令集, 还有自己特有的命令。本章将给出普遍被支持的 AT 命令集, 该命令集是必不可少的。

2.2.1 Modem 状态

在 AT 命令建立之前, DTE 与 Modem 之间是通过硬件电路来直接控制的。RS-232C 接口中的 TxD 和 RxD 仅仅用于发送和接收数据。这使得 DTE 一方面需要完成许多与线路有关的工作, 例如振铃检测等; 另一方面, 随着技术的发展, Modem 要能自动选择合适的传输速率, 而 DTE 需要知道当前的通信速率, 有限的硬件信号很难适应 Modem 的进一步发展。

AT 命令的建立使得 DTE 从以上杂务中解脱出来, 振铃检测、载波检测、速率选择等都可以由 Modem 完成。此时 TxD 和 RxD 不仅仅是传输数据, 还传送 AT 命令。DTE 发送 AT 命令到 Modem, Modem 执行后通过 RxD 返回结果给 DTE。这些命令和返回结果码符合 RS-232C 数据格式。Modem 通过 AT 命令和几个特定的 S 寄存器的状态确定 DTE 发出的是命令还是数据。

与此相对应, Modem 的状态也可以分为命令状态和在线状态。除了拨号占据短暂的时间之外, Modem 总是处于其中一种状态, 当 Modem 启动后, 首先处于命令状态, 连接后进入在线状态。在命令状态下 Modem 以 AT 命令形式接受命令, 例如指示 Modem 去拨号或当电话响铃时给予自动应答。在在线状态下, Modem 与远端系统通信, 这时 Modem 不再尝试对发送给它的数据进行解释, 而是直接将其发送出去。这两种状态可以相互转换。

1. 命令状态

当 Modem 处于命令状态时, Modem 不是和远端系统通信, 而是准备接收命令。此时 Modem 一般都处于挂机状态 (离线命令状态), 但也可以处于摘机状态 (在线命令状态)。Modem 接收并解释 AT 命令。

只有在命令状态下 DTE 才能对 Modem 进行控制, 包括修改参数和拨号等。

当 Modem 上电时都处于离线命令状态，连接建立后进入在线状态。此时如果 Modem 接收到换码序列+++，则进入在线命令状态。

2. 在线状态

在线状态又称为联机状态或数据状态。当通信双方握手完成，建立通信链路后，Modem 就可以发送和接收数据，此时 Modem 的状态称为在线状态。从电话线来的数据被传送到 DTE。在该状态，Modem 不再对接收到的数据分析处理，因此如果此时发送端发送以 AT 开始的字符串，发送端的 Modem 不认为是 AT 命令，因而不会对该序列进行处理，而只是作为一般的数据发送。当载波信息消失后，Modem 自动返回到命令状态。

3. 状态转换

命令状态和在线状态可以进行相互转换。

(1) 离线命令状态→在线状态

当建立连接后，Modem 由离线命令状态转为在线状态。

(2) 在线状态→在线命令状态

在通信过程中，如果 DTE 向 Modem 发送换码序列+++，在发送前后均有 1 秒钟的保护时间（即 Modem 空闲），Modem 将由在线状态转为在线命令状态。

(3) 在线命令状态→在线状态

DTE 向 Modem 发送 ATO 命令，Modem 将从在线命令状态重新进入在线状态。

(4) 在线命令状态→离线命令状态

如果 DTE 向 Modem 发送 ATH 命令，Modem 将挂机，从而由在线命令状态转为离线命令状态。

(5) 在线状态→离线命令状态

如果由于远端 Modem 挂机或线路中断等原因导致载波信号丢失，那么 Modem 将由在线状态转为离线命令状态。

2.2.2 AT 命令

所有的 Modem 命令都从一个特定的“命令前缀”开始，到一个“命令结束标志”结束。命令前缀通常总是由 AT 两个字符组成，它是 Attention 的缩写，意思是“引起注意”，因而称 Modem 命令为 AT 命令。命令结束标志是一个单字符，其值存放在寄存器 S3（具体请参见下节的叙述）中，通常为回车符<CR>。S3 中的内容可以用 AT 命令修改，因而命令结束标志是可以改变的。

命令行除非特殊情况一般均由 AT 或 at 开始，而不是 At 或 aT。Modem 从这两个字符中能检测出波特率、字长和奇偶校验。当然必须事先设置好计算机的串行口的这些参数。

每个 AT 命令以一个单字符或一个&号后跟一个字符定义组成。基本上所有命令的后继参数均采用十进制形式。每个 AT 命令行可以包括许多条 AT 命令，而只需一个 AT 引导。一个命令行被正确执行的前提是其中每个命令都是正确的，否则将被丢弃。执行完命令行后，Modem 并不清除命令缓冲区，而一直保存到下一条新的 AT 命令到来为止。惟一没有命令前缀和结束标志的命令是 A/，它用于告诉 Modem 重复执行存于缓冲区中的上个命令行。

每当 Modem 执行完一个 AT 命令之后，Modem 都会返回结果码以对接收到的命令作出响应。在 Windows 98 界面中，启动“开始→程序→附件→通讯→超级终端”，会出现超级终端窗口，超级终端的一个实际操作结果如图 2.4 所示。



图 2.4 超级终端中的 AT 命令操作实例

为了讨论和实际应用的方便，笔者将 AT 命令按功能分类成 13 个组。这 13 个组分别为：

- (1) 用户接口命令
- (2) 拨号呼叫
- (3) 应答呼叫
- (4) 专线方式
- (5) 状态切换命令
- (6) 挂机命令
- (7) 逻辑接口命令
- (8) 扬声器控制
- (9) 版本信息及自检测试
- (10) 配置命令
- (11) S 寄存器操作
- (12) 连接性选择命令
- (13) 其他通用命令

下面给出的命令基本上是所有的 Modem 都能识别的命令。对于具体 Modem，请参见用户使用说明书。

1. 用户接口命令

当 Modem 处于命令状态时，每发送一个 AT 命令，Modem 都会以一个结果码（通常为 OK 或 ERROR）响应以指示当前命令执行情况。用户接口命令用于完成对 AT 命令的回显和结果码控制。

注意：下面给出的 AT 命令后面的 n 表示命令的参数，许多命令的 n 都具有默认值。例如 E 命令，其默认值为 1，这意味着“ATE1”命令串与“ATE”命令串的功能相同。

□ En

功能为命令回显。

该命令控制 Modem 在命令状态打开或关闭 AT 命令回显。一旦 Modem 进入在线状态，则所有输入均作为数据处理。例如发送 ATZ 命令，当打开回显时，DTE 将收到“ATZ”这 3 个字符；而在关闭回显时，DTE 将无法收到“ATZ”这 3 个字符。

n=0 禁用本地回显。

n=1 启动本地回显（默认值）。

□ Vn

功能为结果码格式。

该命令选择 Modem 返回给 DTE 的结果码是数字形式符还是字符形式符。例如 ATMO 命令被正确执行后，将返回 0（数字码）或 OK（字符码）。各结果码对应的数字形式与字符形式详见表 2.3。由于单个数字比较容易识别，因此在程序设计中常选择数字响应方式，但它被错误识别的可能性也将增大。

n=0 结果代码以数字形式发送（短格式或数字）。

n=1 结果代码以单词形式发送（长文本格式或冗余格式）（默认值）

□ Qn

功能为结果码控制。

该命令控制 Modem 是否发送结果码到 DTE。

n=0 启用结果代码（默认值）。

n=1 禁用返回结果代码（静噪）。

□ Xn

功能为结束码类型 / 呼叫进程。

Xn 结果代码设置/呼叫进度选项。Xn 选择结果代码集和拨号功能。Vn 命令决定结果代码是以单词还是以数字形式发送。

参数：n = 0~7（视 Modem 的型号而定）

n=0 启用连接（CONNECT）结果代码，禁用 CONNECT XXXX 结果代码。不检测占用信号和拨号音频。

n=1 Modem 进入盲拨号状态，启用 CONNECT XXXX 结果代码。不检测占用信号和拨号音频。

n=2 Modem 在拨号前等待拨号音频，启用 CONNECT XXXX 结果代码。不检测占用信号。

n=3 Modem 进入盲拨号状态，启用 CONNECT XXXX 结果代码。如果检测到占用信号，Modem 发出占用（BUSY）结果代码。

n=4 Modem 在拨号前等待拨号音频，启用 CONNECT XXXX 结果代码。如果检测到占用信号，Modem 发出占用（BUSY）结果代码（默认值）。

2. 拨号呼叫

拨号呼叫命令用于完成一个 Modem 向另一个 Modem 发起呼叫, 并建立通信链路的任务。连接成功之后, Modem 通常由命令状态转入在线状态。

□ Dn

功能为拨号命令。

该命令使 Modem 立即进入摘机状态, 并拨出随后的号码(拨号串)以试图建立连接。如果 D 命令后面没有跟拨号串, 则 Modem 将进入在线状态, 并确认在呼叫模式。

D 命令是基本的拨号命令, 它受到其他一些命令(如 T 命令、P 命令)的影响。脉冲拨号的情况下, 非数字字符是无效的。拨号串由拨号修饰符构成。拨号修饰符用于指示 Modem 何时拨号以及如何拨号等操作。

拨号修饰符的说明如下。在拨号串中可以加入连字号(-)和括号等标记字符, 它通常加在数字 0~9 和空格之间, 用于提高命令行的可读性。例如: ATDT(0571)7931999 或 ATDT0571-7931999, 其中 0571 表示长途区号, 7931999 为本地电话号码, 增加可读性。

拨号后, Modem 就一直等待对方 Modem 发来的载波信号, 如果在指定的时间(由 S7 寄存器指定)内没有检测到载波, 则 Modem 将自动释放线路, 返回 NO CARRIER 结果码给 DTE。如果检测到载波, 则通信双方进行差错控制的协商(例如: CARRIER 14400、PROTOCOL: LAPM 等), 最后返回 CONNECT 9600(或其他)等表示连接速率的结果码, 表示连接成功。此时 Modem 自动进入在线状态(除非拨号中有“;”修饰符), 双方可以开始进行通信。

(1) 脉冲拨号方式

P 命令 Modem 使用脉冲拨号, 直到选用音频拨号(T)为止。P 可理解为 Pulse(脉冲)的简写。脉冲的拨号/间隔比率由 &P 命令选择。在遇到某些国家不设脉冲拨号的情况下, P 命令将被忽略。

(2) 音频拨号方式

T 命令 Modem 使用音频拨号, 直到选用脉冲拨号(P)为止。T 可理解为 Tone(音频)的简写。音频号的持续和间隔时间由寄存器 S11 设置。

(3) 长时间暂停

W 命令 Modem 暂停, 直到检测到第二次拨号音频, 检测到拨号音频后, 立即开始拨电话号码。在通过 PBX 拨号或使用某些长途电话服务的情况下, 这可能会很有用。最长等待时间在寄存器 S7 中设置。

例如: ATDT9W7931999

(4) 重拨上次的号码

L 命令 Modem 重拨自开机后所拨的最后一个号码。这应是 ATD 后的第一条命令, 否则 Modem 将忽略该字符。

(5) 拨号后返回命令状态

分号(;)强制 Modem 在拨号后联络不断开的情况下保持在命令方式。分号必须放在拨号命令的结尾。

(6) 等待无声回答

@字符使 Modem 在处理拨号串的下一个符号前在静噪数秒后寻找响铃。S7 寄存器决定

最长等待时间。如果检测到静噪应答，将执行此命令后的拨号修改程序。如果检测到占用信号，Modem 返回占用 (BUSY) 结果代码并进入挂断过程，同时中止命令的进一步执行。

(7) 挂机闪烁

感叹号 (!) 使 Modem 处于挂机状态 0.5 秒，然后返回摘机状态，某些 PBX 系统用此命令来访问诸如呼叫转传和呼叫转送等特殊功能。

(8) 标准暂停

逗号 (,) 使 Modem 在拨号期间暂停一段指定时间，持续时间由寄存器 S8 设置。

目前许多单位 (如宾馆，学校) 都装有内部电话，当这些电话拨市话或长途时通常需要先拨 0 (或 8、6 等)，等听到二次拨号 (外线) 之后才能再拨后续电话号码。此时可以使用逗号 (,) 迫使 Modem 作简短暂停。默认时暂停时间为 2 秒，它由 S8 寄存器指定。

例如: ATDT9,5738686

用 201 电话卡拨打电话时，用 ATDT201,1,79853332#,785633#,05717931666#

(9) 拨号数字和字符

0 到 9 为脉冲或音频拨号的有效数字。

A 到 D、#、* 为拨号字符，仅为音频拨号的有效字符。A、B、C 和 D 是双音频多频节率 (DTMF) 系统中加在 369# 键右边的 4 个键。当有些国家禁用这些字符时，它们将被忽略。连字号 (-) 和括号用作格式化拨号串，不起作用，可忽略。

(10) 使用预先存储的号码拨号

对于要经常使用的电话号码，可以使用 AT&Zn=x (其中 n=0~3 表示存储位置，x 表示电话号码) 命令将电话号码保存到 Modem 的 NVRAM 中。

例如: AT&Z1=7932173 //将电话号码 7932173 保存到位置 1

AIDS1 //拨预存储在位置 1 中的电话号码 7932173

注意: 电话号码 (拨号串) 输入时可包含空格或其他标点符号。对 T 和 P 进行修改的命令可出现在拨号串的任何地方。因此，如果所在的国家允许使用此功能，信号发送方法可能会在传送了几个数字后发生改变。

3. 应答呼叫

当 Modem 检测到远端系统来的一个呼叫时，电话铃声响，此时 Modem 有两种方式可用于完成应答：手工应答和自动应答。

□ A

功能为手工应答。

A 命令使得 Modem 立即摘机，并等待来自远端 Modem 的拨号呼叫和载波信号，试图应答呼叫，而不需等待呼叫振铃信号。对于一个应答，Modem 等待 S7 寄存器所指定的时间，如果没有检测到载波信号，则自动挂机。A 命令适合在手动应答呼叫或在发送方式下直接与另一台 Modem 建立联络时使用。

注意: 在同一命令行上 A 后的任何命令均会被忽略，某些国家可能不允许使用手动应答呼叫 (用 A 命令)。

□ S0=r

功能为自动应答。

如果要求 Modem 具有自动应答特性, 则应该预先将 Modem 的 S0 寄存器设置为非 0 值。

例如: ATSO=r

其中 r 表示在响铃 r 次之后 Modem 自动摘机并试图连接, r 值范围为 1~255, 通常设置为 1。只要 S0 为非 0, 则 Modem 便具有自动应答特性; 如果设置 r 为 0, 则禁止自动应答。当自动应答被禁止后, 则每次电话铃响时, Modem 返回 RING 结果码, 但不应答呼叫, 除非此时执行 ATA 命令。

4. 专线方式

电话网络分为两种模式: 拨号线和专线。

拨号线 (Dial Line) 是指以一个专门号码作为拨号连接计算机和终端的通信线路, 数据通过电话线路传送到计算机, 话路质量相对较差、数据传输不稳定、保密性较差。

专线 (Leased Line) 是指在一台计算机和另一台终端间的永久通信连接, 完全由租用者支配。两端点固定不变, 保密性强, 线路质量好, 数据传输稳定、可靠、高效。如果利用专线进行声音通信, 则不必拨号便可以通话。当使用专线时, 两边电话同时进入摘机状态, 一旦摘机便可以开始通话。

如果采用专线方式通信, 首先应将 Modem 工作状态设置为专线连接方式。部分 Modem 采用硬件开关设置, 而绝大多数 Modem 都提供专线连接的 AT 命令 &L。

□ &Ln

功能为设置专用线路操作。

该命令用于设置 Modem 线路方式。

n=0 选择拨号线操作 (默认值)。

n=1 选择专线操作。

如果双方使用 Modem 建立通信链路, 则可以使用 AT 命令建立连接。加入的 Modem 能配置成自动建立载波信号, 这样通过专线连接的两个 Modem 只需执行了 AT&L1 命令, 就自动开始建立连接。如果 Modem 不具有自动建立载波信号的功能, 则可以通过前面“背靠背连接”中介绍的方法建立连接, 即一方使用 ATD 命令摘机呼叫, 另一方使用 ATA 命令响应, 专线也就相当于“调制电缆”。

5. 状态切换命令

当通信双方建立通信链路之后, 就从命令状态进入在线状态, 此时双方可通过电话线发送和接收数据。命令状态和在线状态之间的转换可以通过 AT 命令完成。

□ +++

功能为从在线状态切换到在线命令状态。

+++ 为退出代码序列, 是一个换码序列 (转义序列), 而不是一个 AT 命令, 因而前面不加 AT, 后面也不用跟回车符。寄存器 S2 中的字符集以极快的速度连续三次发送给 Modem, Modem 暂时退回到命令方式。退出字符的默认值为+, 说明文件中如果指明要输入+++, 迅速连续三次输入寄存器 S2 中字符。退出代码序列不要以 AT 开头, 输完后也不要按 Enter 键。当要返回联机方式时, 请使用 ATO 命令。

例如: 1 秒+++1 秒

□ On

功能为从在线命令状态返回到在线状态。

On 将 Modem 强制为联机方式。

参数: $n=0$ 、1、3 (视调制解调器的型号而定)

$n=0$ 进入联机方式。

$n=1$ 进入联机方式, 并初始化均衡器重整。

$n=3$ 进入联机方式, 并在返回联机数据方式前发出通信速率重协商。

例如: AT0

6. 挂机命令

在通信结束后, 应挂机、拆除线路。

□ Hn

功能为挂机/摘机控制。

Hn 控制挂机, 延迟。

$n=0$ Modem 挂机 (挂断) (默认值)。

$n=1$ Modem 摘机。

注意: H1 在某些国家可能不允许使用。在那种情况下, ATH1 将返回一个错误代码。

例如: +++ //首先切换到命令状态

ATH //然后挂机

□ Yn

功能为长空挂断。

该命令允许或禁止长空挂断(Long Space Disconnect)。长空挂断可用于挂断两个 Modem, 一般情况下不使用该命令挂机, 除非有一个 Modem 处于很远的一个无人看管的地方, 而且需要通过控制本地的 Modem 来获得对远端 Modem 的控制。

$n=0$ 禁止长空挂断 (默认值)。

$n=1$ 允许长空挂断。

长空挂断是指当检测到任何挂断状态(DTR 由 ON 跳变为 OFF)时, Modem 立即发送一个 4 秒的空信号给对方的 Modem, 该信号通知对方 Modem 挂断。在另一端, 如果已执行 ATYI 命令, 即允许长空挂断, 则只要接收到 1.6 秒的空号, Modem 将丢失载波, 然后挂断。这可能引起无意的挂断, 因此大多数 Modem 都默认禁止长空挂断。

□ Zn

功能为软件复位/恢复保存的预置文件。

Modem 执行一个软件复位并恢复指定的预置配置文件命令, Z 命令后面的所有命令将被忽略。Modem 执行一个复位命令之后, 将返回到电源开启时的状态, 并在各个变量中重新装入其默认值, 同时从 NVRAM (非易失存储器) 中读取预置文件, 最后进行自检。因此如果 Modem 处于在线命令状态, 该命令将使 Modem 自动挂机。

$n=0$ 软件复位并恢复预置文件 0 (默认值)。

$n=1$ 软件复位并恢复预置文件 1。

7. 逻辑接口命令

Modem 提供一组 AT 命令，它规定了 Modem 逻辑接口（或 RS-232C 接口）的行为。

□ &Cn

功能为数据载波检测（DCD）选择。

AT&Cn 控制 DCD 选项。

n=0 DCD 总处于打开（ON）状态，来自远端 Modem 的数据载波状态被忽略。

n=1 检测到数据载波时，DCD 打开（ON）；未检测到数据载波时 DCD 关闭（OFF）（默认值）。

□ &Dn

功能为数据终端准备就绪（DTR）选择。

AT&Dn 控制 DTR 选项。

n=0 Modem 忽略 DTR（默认值）。

n=1 Modem 在检测到 DTR 由打开向关闭转换时，进入命令方式。

n=2 Modem 挂断，进入命令方式，在检测到 DTR 由打开向关闭转换时禁用自动应答。

n=3 Modem 在检测到 DTR 由打开向关闭转换时，进入初始化状态。

□ &Rn

功能为请求发送/清除发送（RTS/CTS）选择。

该命令控制由 Modem 到 DTE 的 CTS 信号的操作。

n=0 当 Modem 在线时，CTS 跟随 RTS 的变化（默认值）。

n=1 当 Modem 在线时，CTS 一直有效（ON），忽略 RTS 信号。

□ &Sn

功能为数据设备就绪（DSR）选择。

该命令控制由 Modem 到 DTE 的 DSR 信号的操作。DSR 指示 Modem 是否已经准备好。

n=0 DSR 一直有效（ON）（默认值）。

n=1 只有在握手时 DSR 有效，即检测到载波信号时允许 DSR，失去载波信号时禁止 DSR。

8. 扬声器控制

在拨号过程和数据传送过程中，可通过编程控制 Modem 的扬声器音量大小以及开关状态。

□ Ln

功能为扬声器音量控制。

使用 Ln 控制在传真和数据通信期间扬声器的音量。

n=0 扬声器低音量。

n=1 扬声器低音量。

n=2 扬声器中音量（默认值）。

n=3 扬声器高音量。

注意：要彻底关闭扬声器，请使用 M0 命令。

□ Mn

功能为扬声器开关控制。

使用 Mn 控制在传真和数据通信期间扬声器的开/关。

n=0 扬声器关闭。

n=1 扬声器打开，直到检测到载波（默认值）。

n=2 调制解调器在摘机状态时扬声器一直打开。

n=3 拨号后扬声器打开，直到检测到载波。

9. 版本信息及自检测试

每一个 Modem 中都存有版本信息，包括 Modem 的型号、制造厂家、校验和等信息，另外可通过自检命令对 Modem 进行测试。

□ In

功能为 Modem 版本信息。

In 向 Modem 询问其产品识别代码，ROM 校验和或 ROM 校验和状态。

n=0 返回产品版本（默认值）。

n=1 计算并显示 ROM 校验和（如 12AB）。

n=2 执行 ROM 校验，计算并核对校验和，显示确定或错误。

n=4 返回数据激励的软件版本。

n=9 返回国家代码。

□ &Tn

功能为自检命令。

该命令用来建立或终止回送检测，以检测 Modem 与 Modem 之间、Modem 与 DTE 之间通信的完整性。AT&Tn 选择 8 条测试命令中的一条。

n=0 终止任何进行中的测试（默认值）。

n=1 初始化本地模拟环回（ALB），如果正在进行呼叫，将返回一条错误信息。

n=3 本地数字回送。

n=4 同意远端数字回送请求。

n=5 禁止远端数字回送请求。

n=6 远端数字回送。

n=7 带自检的远端数字回送。

n=8 带自检的本地数字回送。

10. 配置命令

Modem 提供一组命令用于 Modem 的配置管理。Modem 一般提供 3 种配置：厂家配置（Factory Profile）、当前配置（Active Profile）和存储配置（Stored Profile）。其中厂家配置存放在 Modem 的 ROM（只读存储器）中，当前配置存放在 Modem 的 RAM（可读可写存储器）中，存储配置存放在 NVRAM（非易失存储器）中。

□ &Fn

功能为恢复厂家配置。

Modem 重新调出“厂家原始配置”作为当前 Modem 的活动配置，但不保存该配置。如

果要保存该配置, 可使用 AT&W 命令。

n=0 恢复厂家原始配置文件 0 (默认值)。

n=1 恢复厂家原始配置文件 1。

只有部分厂家具有两个厂家配置 0 和 1, 大多数厂家只提供一种厂家配置 0。

□ &Wn

功能为保存当前配置。

将当前 Modem 的活动配置 (当前配置) 保存到由 n (0 或 1) 指定的 NVRAM 中, 包括保存 S 寄存器的当前设置。NVRAM 中的配置不会随着 Modem 的关电而丢失。当再次使用 &W 命令写入新的配置时, 将导致原先的配置被覆盖而丢失。如果 Modem 重新复位或使用 Z 命令对 Modem 软复位, 存储在 NVRAM 中的存储配置将成为当前配置。

n=0 保存当前配置到预置文件 0 (默认值)。

n=1 保存当前配置到预置文件 1。

□ Zn

功能为软复位。

Modem 执行一个软件复位并恢复指定的存储配置文件为当前配置。

n=0 软件复位并恢复预置文件 0 (默认值)。

n=1 软件复位并恢复预置文件 1。

□ &Yn

功能为指定启动配置。

当 Modem 上电或执行 ATZ 命令时, Modem 将自动恢复两个存储配置中的一个作为当前配置。&Y 命令用于指示 Modem 恢复哪个存储配置。

n=0 上电/软件复位时恢复预置文件 0 (默认值)。

n=1 上电/软件复位的恢复预置文件 1。

□ &Vn

功能为描述显示当前配置的和存储的预置文件。

如果要查看当前的和存储的配置文件, 可以使用该命令。

n=0 预置文件 0 (默认值)。

n=1 预置文件 1。

□ &Zn=X

功能为存储电话号码。

Modem 中的 NVRAM 最多可存储 4 个常用的电话号码。该命令将拨号率保存到 NVRAM 中, 可使用 ATDS=n 命令来拨出存储在位置 n 的电话号码。

n=0~3。

X=拨号串。

11. S 寄存器操作

Modem 内部有许多寄存器, 称为 S 寄存器。S 寄存器提供了存取 Modem 设置的专用方法。例如在应答前允许电话振铃的次数等配置信息都由 S 寄存器中的命令指定。通过 AT 命令可以存取 S 寄存器数值。

□ Sn=X

功能为写 S 寄存器。

Sn 将指针指向某个 S 寄存器，其中 n 为寄存器号。在选择另一个寄存器之前，n 值可用 AT? 读出，并可用 AT= 修改。

X=0~255。

□ Sn?

功能为读 S 寄存器。

Sn? 报告由 n 指定的寄存器值，n 可为任何有效的 S 寄存器号。该命令使 Modem 返回 S 寄存器 n 的数值（十进制）。

如果要查看所有 S 寄存器中的内容，可以使用 AT&V 命令。

12. 连接性选择命令

以下 AT 命令影响 Modem 与其他 Modem（或通信标准）相互作用的方式。

□ Bn

功能为 ITU 或 Bell 标准。

参数视调制解调器的型号而定。

n=0 ITU V.22 速率为 1200 bit/s；V.21 速率为 300 bit/s。

n=1 贝尔（Bell）212A 速率为 1200 bit/s（默认值）。

n=2 或 3 调制解调器发送时采用 CCITT V.23 R1200/T75 ASB；
调制解调器接收时采用 CCITT V.23 T1200/R75。

n=15 V.21 速率为 300 bit/s。

n=16 贝尔（Bell）103 速率为 300 bit/s。

□ &Gn

功能为选择校正音。

该命令只适用于 1200bit/s（V.22）和 2400bit/s（V.22bis）的连接。

n=0 禁止校正音（默认值）。

n=1 允许 550Hz 校正音。

n=2 允许 1800Hz 校正音。

□ &Mn

功能为异步通信方式。

n=0 选择 Modem 工作于异步通信工作方式。

□ &Qn

功能为异步通信方式。

n=0 异步方式。

n=5 错误控制方式（默认值）。

n=6 异步方式。

13. 其他通用命令

下面给出其他一些通用 AT 命令。

□ A/

功能为重复上一条命令。

Modem 重复上一次的 AT 命令。它主要用于重拨电话号码。当第一次拨号因线路忙或无应答等造成的连接失败时,使用该命令尝试第二次拨号呼叫。该命令前面不用加 AT,后面也不用跟回车符。

□ &Kn

功能为流量控制。

命令定义 Modem 和 DTE 间的流量控制(流控)方式。

n=0 禁止流控。

n=3 允许 RTS/CTS 流控(默认值)。

n=4 允许 XON/XOFF 流控。

n=6 允许 RTS/CTS 和 XON/XOFF 两种流控。

□ Nn

功能为调制信息交换。

在本地调制解调器和远端调制解调器连接时,如果两者通信速度不同,可使用 Nn 控制本地调制解调器是否执行协商信号交换。

n=0 发送或应答时,交换仅采用由 S37 和 ATB 命令指定的通信标准。

n=1 发送或应答时,采用由 S37 和 ATB 命令指定的通信标准开始交换,交换期间速度可能会降低(默认值)。

2.2.3 S 寄存器

为了满足用户对 Modem 的一些高级操作和控制的要求, Hayes 公司将某些 Modem 参数保存到 RAM 内的寄存器中,并称此寄存器为 S 寄存器。S 寄存器提供了存取 Modem 设置的专用方法以及控制 Modem 工作的高级技术。

DTE 可通过 ATSn? 和 ATSn=X 命令读写 S 寄存器。例如:设置 Modem 在振铃一次后自动应答远端 Modem 的呼叫,则键入 AT\$0=1。

例如: AT\$0=1 设定 S0 为 1。

AT\$0? 读 S0 内容。

AT\$0=0 设定 S0 为 0。

AT? 读最后存取(读或取)寄存器。

不同型号的 Modem,其 S 寄存器的个数和作用都有所不同。如同 AT 命令一样,下面仅列出最常用的 S 寄存器。对于具体 Modem,请参见用户使用说明书。

□ S0 寄存器

范围: 0~255

单位: 振铃次数

功能为在应答前允许电话振铃的次数。

该寄存器表示 Modem 开始自动应答前需要等待的振铃次数。默认值为 0,表示禁止 Modem 的自动应答特性。如果线路只用于 Modem 通信,则通常将 S0 设置为 1。如果线路还用作话音通信,则应将 S0 设置为较大的数值。

□ S1 寄存器

范围：0~255

单位：振铃次数

功能为振铃计数。

该寄存器表示电话已振铃的次数，每响一次加一，振铃结束后自动清零。它是一个只读寄存器，主要用于 Modem 内部工作，用户一般无需使用。默认值为 0。

□ S2 寄存器

范围：0~127

单位：十进制 ASCII 值

功能为换码字符。

该寄存器含有表示由在线状态到命令状态的换码字符，默认值为 43。默认字符为“+”（ASCII 值为 43）。连续 3 个换码字符就构成一个换码序列，完成由命令状态到在线状态的切换。

□ S3 寄存器

范围：0~127

单位：十进制 ASCII 值

功能为回车字符。

该寄存器定义了回车字符的值，默认为通常的回车符“CR”（ASCII 值为 0DH），默认值为 13。Modem 将回车字符作为 AT 命令行的结束并且作为结果码的结束符发送。

□ S4 寄存器

范围：0~127

单位：十进制 ASCII 值

功能为定义换行字符的值。

该寄存器默认为通常的换行符“LF”（ASCII 值为 0AH），默认值为 10。Modem 将换行字符作为一个回送命令行结束符或一个结果码的结束符发送。

□ S5 寄存器

范围：0~32, 127

单位：十进制 ASCII 值

功能为定义回退字符的值。

该寄存器默认为通常的退格符“BS”（ASCII 值为 08H），默认值为 8。回退字符完成退格操作。

□ S6 寄存器

范围：2~255

单位：秒

功能为定义 Modem 从摘机到开始拨号之间应等待的时间。

该寄存器默认值为 2。

□ S7 寄存器

范围：1~255

单位：秒

功能为定义等待载波信号的时间。

该寄存器定义了拨号后或者应答一个呼叫时, Modem 等待载波信号的最大时间, 默认值为 50。如果在这段时间内没有收到远端 Modem 的载波信号, 则 Modem 将挂机并且返回 NO CARRIER 结果码。对于远距离或国际呼叫时, 应该将该寄存器设置为较大数值。

□ S8 寄存器

范围: 0~255

单位: 秒

功能为决定了当 Modem 在拨号串中遇到逗号 (,) 时应暂停的时间。

该寄存器默认值为 2。

□ S9 寄存器

范围: 1~255

单位: 0.1 秒

功能为指定 Modem 识别远端 Modem 发来的载波信号, 并确认载波信号的时间长度。

如果时间过短, 则有可能将干扰误认为载波信号, 默认值为 6。

□ S10 寄存器

范围: 1~255

单位: 0.1 秒

功能为指定从丢失载波信号到挂机的时间。

该寄存器指定了从 Modem 发现远端载波信号丢失到挂机之间的时间。该延时允许 Modem 忽略短暂的中断而拆除连接。如果该设备值小于 S9 中的设置值, 则一旦失去载波信号将立即导致挂机并拆除连接。

□ S11 寄存器

默认值为 95。

范围: 50~255

单位: 0.001 秒 (1 毫秒)

功能为指定按键拨号音频的持续时间。

拨号速率是该寄存器值倒数的二分之一。因此当使用 95 毫秒默认值时, 将产生大约每秒 $(1000/95)/2=5$ 个数字的拨号速率。

□ S12 寄存器

默认值为 50。

范围: 0~255

单位: 0.02 秒

功能为指定 Modem 在换码序列开始之前和之后必须空闲的时间。

该寄存器默认为 $50 \times 0.02 = 1$ 秒, 因此 “1 秒+++1 秒” 将导致 Modem 由在线状态切换到命令状态。如果换码序列之前或之后的时间小于 1 秒, 则 “+++” 将当作普通数据发送, 而不认为是换码序列。

□ S37 寄存器

默认值为 0。

范围: 0~12、...

单位：无

功能为决定最高 DCE 速率。

该寄存器决定了 Modem 与远端 Modem 建立连接时的最高线路速率。经过协商，最后的线路速率为通信双方都支持的速率。ATNn 命令控制自动速率的检测。具体 S37 寄存器数值对应的线路速率随不同 Modem 而不同，因此请用户参见所使用的 Modem 说明书。

2.2.4 Modem 返回信息码

当 Modem 处于命令状态时，每当 PC 机发送一条 AT 命令后，Modem 至少返回一个结果码，以指示当前命令是否正确执行以及执行结束。

结果码有两种形式：一种是文本信息（即字符串形式），另一种是数字码。如果程序员自编软件来控制 Modem，则最好使用数字码形式，它的优点是软件处理结果码简单方便。如果使用 Modem 的附带软件用键盘来控制 Modem，则可以使用字符串形式，它的优点是结果清楚明了。

AT 命令集中提供了 V 命令来选择结果码返回形式。ATV0 命令表示以数字形式返回结果码，ATV1 命令表示以字符串形式返回结果码。

随着 Modem 的速率的提高和功能的增强，结果码随之增加。但各厂家所定义的结果码并不统一。表 2.3 列出大多数 Modem 都支持的结果码，而对于每个具体的 Modem 结果码可能有所不同，详细情况请参考 Modem 用户手册。

表 2.3 Modem 结果码

结果码	含义
OK	命令正确执行
CONNECT	连接建立
RING	检测到振铃信号
NO CARRIER	没有接收到载波信号或载波信号丢失
ERROR	无效命令、命令行错误、或错误超过 255 个字符
CONNECT 1200	在 1200 bit/s 速度下建立连接
NO DIALTONE	没检测到拨号音(X2、X4 命令、W 生成的)
BUSY	检测到忙音(X3、X4 命令生成的)
NO ANSWER	当拨一个不提供拨号音的系统时，检测不到无声信号(@生成的)
CONNECT 2400	2400 bit/s 速率下建立连接(由 X0 命令关闭)
CONNECT 4800	4800 bit/s 速率下建立连接(由 X0 命令关闭)
CONNECT 9600	9600 bit/s 速率下建立连接(由 X0 命令关闭)
CONNECT 19200	19200 bit/s 速率下建立连接(由 X0 命令关闭)

2.3 通用异步接收发送器 UART 概述

随着大规模集成电路的出现, 串行通信接口的功能也扩大了, 它不但能实现异步、同步通信, 而且还能实现各种通信规程中的功能, 较常用的是 SDLC/HDLC 规程。

目前, 许多厂家都提供大规模集成电路的可编程串行通信接口芯片, 可用于同步通信方式或异步通信方式。通用可编程的同步、异步通信接口芯片统称为 UART (Universal Asynchronous Receiver/Transmitter, 通用异步接收/发送装置)。UART 既能实现异步、同步通信方式, 也能实现 SDLC / HDLC 规程方式。

模拟 Modem 利用 UART 来进行串行通信, UART 是一个将并行输入转换为串行输出的芯片, 通常集成在主板上, 多数是 16550AFN 芯片。因为计算机内部采用并行数据, 不能直接把数据发送到 Modem, 必须经过 UART 整理才能进行异步传输, 其过程为: CPU 先把准备写入串行设备的数据放到 UART 的寄存器(临时内存块)中, 再通过 FIFO (First Input First Output, 先入先出队列)传送到串行设备, 若是没有 FIFO, 信息会杂乱无章, 无法被传送到 Modem。

1. UART 的结构原理

UART 接口芯片能同时进行发送和接收, 叫做双工方式。它具有发送器和接收器的功能, 并且还有控制部件, 以完成各种控制功能。

CPU 是通过并行数据线将数据输入发送器的。在采用异步通信方式时, 发送器接收 CPU 发送来的数据位后, 要附加上起始位、奇偶位及停止位形成异步通信信息格式, 然后按发送时钟节拍, 通过移位寄存器将并行码串行地发送出去。在采用同步通信方式时, 发送器要把 CPU 送来的数据段附加上同步字及监督位, 形成同步通信信息格式, 通过移位寄存器串行地发送出去。发送器是通过一根串行数据输出线 TxD 输出数据的。

接收器是通过一根串行数据输入线 RxD 接收输入数据的。接收器在异步通信方式时, 始终监视着 RxD 线, 当发现一个起始位时, 就按约定的接收时钟频率接收数据, 由 RxD 来的串行数据先进入移位寄存器, 去掉起始位、停止位, 检查有无奇偶错误, 然后并行输入给接收数据缓存器(变为并行数据), 以便 CPU 用输入指令取走数据。在同步通信方式时, 接收器在收到同步字符时将其作为接收数据段的开始, 接收完数据后, 要检查伴随式是否为零, 以确定是否发生了错码。然后并行输入给接收数据缓存器, 以便由 CPU 取走。

控制部件是用来控制 UART 的工作的, CPU 通过它进行芯片内的工作控制, 并且用它来向外部设备发出控制命令, 或从外设读取状态。

2. UART 的编程

UART 的功能可通过编程来选择和确定, 这叫做初始化工作。编程的内容主要有以下几个方面:

(1) 确定工作方式, 例如用同步通信方式, 还是用异步通信方式。

(2) 在异步工作方式时, 要确定接收时钟频率与波特率的比例系数。例如采用波特率系数是 16 倍、32 倍还是 64 倍, 必须预先选定。

在编写操作 UART 的汇编程序时，请读者参见 UART 的相关资料。

本章小结

本章介绍了串行通信标准中的 RS-232C 标准，包括引脚定义、分类、标准、安装和使用。接着介绍了 PC 机 Modem 领域的 Hayes 标准，分为 3 部分介绍，即首先介绍 Modem 的命令状态和在线状态两种模式以及它们之间的相互转换；然后详细介绍 AT 命令集，AT 命令是 Hayes 标准的核心，程序只有通过发送 AT 命令才能控制 Modem；最后介绍 Modem 的 S 寄存器，S 寄存器的操作是 Modem 控制的高级技术。最后简单介绍了通用异步接收发送器 UART。

第3章 嵌入式汇编语言开发通信程序

本章主要内容:

- Delphi 中的嵌入式汇编语言基础知识
- 嵌入式汇编语言的通信编程例子

汇编语言程序的编译效率和执行速度都很高,用汇编语言直接对串口进行操作对弥补串行通信速度较慢的缺陷有好处。具体做法是:用汇编语言编写读、写串口的函数,在通信程序中直接调用这些函数。

Delphi 允许在 Object Pascal 程序中直接编写汇编代码,它可以支持绝大部分的 Turbo Assembler 和 Microsoft Assemble 语法集合,支持所有 8086/8087 和 80386/80387 操作代码,支持 Turbo Assembler 中的几个表达式操作。

除了 DB、DW 和 DD (定义 Byte、Word 和 Double Word) 外,Delphi 的内嵌汇编器不支持 Turbo Assembler 指令(例如 EQU、PROC、STRUC、SEGMENT 和 MACRO 等指令)。

通过 Turbo Assembler 指令实现的大部分操作可以对应到 Object Pascal 指令。例如,大多数 EQU 指令对应于 Delphi 的 constant、variable 和 type 声明,PROC 命令可以对应于 Procedure 和 Function 声明,STRUC 指令对应于 RWecond 类型。

本章首先介绍 Delphi 中的嵌入式汇编语句的语法和指令、表达式元素、汇编过程和函数,然后给出了几个内嵌汇编语句的通信例子。

3.1 Delphi 中的嵌入式汇编语言

Delphi 中可以使用嵌入式汇编语言。语法如下:

asm

statementList

end

statementList 是一个汇编程序语句,其分隔符可以是分号、换行符或者是 Object Pascal 的注释。多条汇编语句可以通过分号分隔放在一行中,如果不在一行中则不用分号。

保留字 inline 和 assembler 指令只是为了保持向后的兼容性而保留了下来,但它们在编译器中没有任何影响。

3.1.1 汇编语句的基础知识

下面介绍汇编语句的语法、寄存器的使用和汇编指令等基础知识。

1. 汇编语句的语法

Label: Prefix Opcode Operand1, Operand2

Label 是个标签, Prefix 是汇编程序的前缀操作码, Opcode 是汇编程序指令操作码或指令, Operand 是汇编表达式。Label 和 Prefix 是可选项。

汇编语句间允许加入注释, 但不是在汇编语句内, 例如:

```
MOV CX,100 {Count}           {正确!}
MOV {Initial value} AX,1;     {错误!}
MOV CX, {Count} 100          {错误!}
```

2. 寄存器的使用

通常, 寄存器在 asm 语句中的使用规则与 external procedure 或 function 的用法相似。asm 语句必须保持 EDI、ESI、ESP、EBP 和 EBX 寄存器的内容, 但可以自由修改 EAX、ECX 和 EDX 寄存器的内容。

在 asm 语句的入口, BP 指向当前的堆栈, SP 指向堆栈顶端, SS 包含了堆栈的段地址, 而 DS 包含了数据段的段地址。除了 EDI、ESI、ESP、EBP 和 EBX 寄存器的内容, asm 语句的入口可以不考虑其他寄存器的内容。

3. 标签

在嵌入式汇编程序中标号的使用方法与 Object Pascal 是相同的——一个标签标识符加一个冒号。标签没有长度限制, 但在内嵌汇编器中只有前 32 个字符有效。与 Object Pascal 一样, 标签必须在包含 asm 语句的块中的标签声明部分定义, 惟一例外的是局部标签的定义, 局部标签必须有 @ 符号开头, 即:

@接一个或几个字母 (A…Z)、数字 (0…9)、下划线 () 或一个 @ 的组合。

例如:

@1

@LOOP

@WAIT_RING

局部标签自动受限制于 asm 语句之中, 并且只能在定义它的 asm 语句之中识别, 也就是说, 一个局部标签的作用范围是从包含该标签的汇编语句的 asm 关键字开始到 end 关键字为止。与一般标签不同的是, 局部标签不需要先在标签说明部分进行说明。

4. RET 指令

RET 指令操作码常常生成一个 near 地址。

5. 自动跳转长度

如果不存在其他定向, 内嵌汇编器将按照最短距离自动优化跳转指令, 从而获得一条跳转指令的最有效形式。该自动跳转的长度适用于无条件跳转指令 (JMP) 和所有的跳转目的的是一个标签而非一个过程或函数的条件跳转指令。

对于无条件跳转指令 (JMP)，如果目的标签的距离在-128~127 字节之内，汇编器生成跳转 (一个字节操作码跟着一个字节偏移)，否则生成一个近跳转 (一个字节操作码跟着两个字节偏移)。

对于条件跳转指令，如果目的标签的距离在-128~127 字节之内，汇编器生成短跳转，否则生成一个带反向条件的短跳转，它跨越一个近地址跳转到目的标签 (总共 5 个字节)，例如：

```
JC      Stop
```

Stop 不在一个短跳转可达的地址范围内，它将自动被转换为如下机器码：

```
JNC     Skip
```

```
JMP     Stop
```

```
Skip:
```

跳转至过程和函数入口总是近跳转。

6. 汇编指令

Delphi 内嵌汇编器支持 3 种汇编指令：DB (定义字节)、DW (定义字) 和 DD (定义双字)。每个生成的数据对应于紧跟其后的由逗号分隔的操作数。

DB 指令产生单字节的数据，每个操作数可以是常量表达式，数值在-128~255 之间，或者是任意长度的字符串。常量表达式产生一个字节的操作数，而字符串产生一序列的字节，其数值顺序对应于字符串中每个字符的 ASCII 码。

DW 指令产生一序列字，每个操作数可以是常量表达式，数值在-32 768~65 535 之间，或者地址表达式。对于地址表达式，内嵌汇编器将产生一个近地址指针，它是一个包括地址偏移部分的字。

DD 指令产生一序列字，每个操作数可以是常量表达式，数值在-2 147 483 648~4 294 967 之间，或者地址表达式。对于地址表达式，内嵌汇编器将产生一个远地址指针，它是一个包括地址偏移部分的字，接着一个包括段地址部分的字。

由 DB、DW 和 DD 指令产生的数据保存在段中，与其他内嵌汇编语句产生的代码相同。要在数据段中定义数据，应该利用 Object Pascal 的 var 或 const 声明。

下面是一些 DB、DW 和 DD 指令的例子。

```
asm
```

DB	0FFH	{一个字节}
DB	0,99	{两个字节}
DB	'A'	{ Ord('A') }
DB	'Hello world...';0DH,0AH	{ CR/LF 跟随的 String }
DB	12,"Delphi"	{ Pascal 风格的 string }
DW	0FFFFH	{一个字 }
DW	0,9999	{两个字}
DW	'A'	{与 DB 'A'、0 相同}
DW	'BA'	{与 DB 'A'、'B'相同}
DW	MyVar	{ MyVar 的 Offset }
DW	MyProc	{ MyProc 的 Offset }

DD	0FFFFFFFFH	{一个双字}
DD	0,999999999	{两个双字}
DD	'A'	{与 DB 'A'、0、0、0 相同}
DD	'DCBA'	{与 DB 'A'、'B'、'C'、'D' 相同}
DD	MyVar	{指向 MyVar }
DD	MyProc	{指向 Pointer to MyProc }

end;

在 Turbo Assembler 中, 当 DB、DW 或 DD 指令之前有一个标识符, 就可以声明一个变量, 例如:

```

ByteVar    DB      ?
WordVar    DW      ?
IntVar     DD      ?
...
MOV        AL,ByteVar
MOV        BX,WordVar
MOV        ECX,IntVar

```

Delphi 的内嵌汇编器不支持变量声明。可以被定义的只有标签, 所有变量必须使用 Object Pascal 语法来声明, 与前面对应有下列语句。

```

var
  ByteVar: Byte;
  WordVar: Word;
  IntVar: Integer;
...
asm
  MOV     AL,ByteVar
  MOV     BX,WordVar
  MOV     ECX,IntVar
end;

```

7. 操作数

内嵌汇编器的操作数可以是常量、寄存器、符号和运算符。

表 3.1 给出了已经预定义了的的操作数, 这些保留字常常优先于用户定义的标识符, 例如:

```

var
  Ch: Char;
...
asm
  MOV     CH, 1
end;

```

表 3.1 内嵌汇编保留字

AH	BX	DI	EBX	ESP	PTR	SS
AL	BYTE	DL	ECX	HIGH	QWORD	ST
AND	CH	DS	EDI	LOW	SHL	TBYTE
AX	CL	DWORD	EDX	MOD	SHR	TYPE
BH	CS	DX	EID	NOT	SI	WORD
BL	CX	EAX	ES	OFFSET	SP	XOR
BP	DH	EBP	ESI	OR		

代码意思为将 1 装入 CH 寄存器，而不是 Ch 变量。要存取一个由用户定义的与保留字同名的符号，必须利用 & 标识符来覆盖保留字。

```
MOV    &Ch, 1
```

注意：最好避免用户定义的标识符与内嵌汇编的保留字同名。

3.1.2 表达式

内嵌汇编表达式是由表达式元素和运算符组成的，每个表达式可以带有一个表达式子句和表达式类型。

1. Object Pascal 与汇编表达式的不同

Object Pascal 与表达式的最大区别在于所有汇编的表达式必须分解成单独的可以在编译时计算的常量。例如，给出如下声明：

```
const
  X = 10;
  Y = 20;
```

```
var
```

```
  Z: Integer;
```

下述则是有效的汇编语句：

```
asm
```

```
  MOV    Z,X+Y
```

```
end;
```

因为 X 和 Y 都是常量，表达式 X+Y 也是一个值为 30 的常量，结果指令就是简单地将立即数 30 送 Z，但是如果将 X 和 Y 改作变量：

```
var
```

```
  X, Y: Integer;
```

这时不能在汇编语句中计算 X+Y，而要利用寄存器进行操作。

```
asm
```

```
  MOV    EAX,X
```

```

ADD    EAX,Y
MOV    Z,EAX
end;

```

在 Object Pascal 中，一个引用的变量被解释成该变量的内容，但在内嵌程序表达式中，引用的是变量的地址。例如，在 Object Pascal 表达式 $X+4$ 中，意思是变量 X 加上 4，而在汇编中的意思是比 X 地址高 4 个字节的地址单元的内容。因此在汇编中甚至允许编写：

```

asm
    MOV    EAX,X+4
end;

```

上面这段代码不是将 $X+4$ 的值赋给 AX ，而是 X 地址加 4 的单元内容送 EAX 。下面的代码将变量 $X+4$ 的值赋给 EAX 。

```

asm
    MOV    EAX,X
    MOV    EAX,4
end;

```

2. 表达式元素

表达式的基本元素是常量、寄存器和符号。

(1) 常量

内嵌汇编器支持两种类型的常量：数值常量和字符串常量。

①数值常量

数值常量必须是整数，汇编器数值必须在 -2 147 483 648 和 4 294 967 295 之间，缺省状态下，数值常量使用十进制，但内嵌汇编器也支持二进制、八进制和十六进制，二进制的表达方法是数值后加 B，八进制的表达方法是数值后加 O，十六进制的表达方法是数值后加 H，或在数值前加 \$。在 Object Pascal 中只允许使用十进制，或 \$ 加数值的方法表达的十六进制。

数值常量必须以一个 0 到 9 的数字或者 \$ 字符开始，当使用 H 后缀的方式表示十六进制数时，如果以 A 到 F 开头，须在数值前添加一个 0，例如：0BZAD4H、\$BAD4。

②字符串常量

字符串必须包括在单引号或双引号之间。例如：

```

'Z'
'Delphi'
"That's all folks"
"That'  's all folks," he said.'
'100'
'''
'''
'''

```

在第四个字符串中利用两个连续的单引号来表示一个单引号字符。

DB 指令中允许任意长度的字符串，并且分配一个对应字符串中各个字符的 ASCII 值的序列。

(2) 寄存器

表 3.2 中标明了 CPU 各寄存器的保留字。

表 3.2 ST 寄存器的保留字

寄存器	保留字
32 位通用寄存器	EAX EBX ECX EDX
32 位指针或变址寄存器	ESP EBP ESI EDI
16 位通用寄存器	AX BX CX DX
16 位指针或变址寄存器	SP BP SI DI
低 8 位寄存器	AL BL CL DL
16 位段寄存器	CS DS SS ES
高 8 位寄存器	AH BH CH DH
协处理器寄存器堆栈	ST

当操作数中仅包括寄存器名称时，则被称作寄存器操作数，所有的寄存器都可以作为寄存器操作数。

基地址寄存器（BX 和 BP）以及变址寄存器（SI 和 DI）可以放在方括号中表示变址。有效的基址/变址寄存器组合是[BX]、[BP]、[SI]、[DI]、[BX+SI]、[BX+DI]、[BP+SI]和[BP+DI]，还可以对所有 32 位寄存器使用变址，例如，[EAX+ECX]、[ESP]和[ESP+EAX+5]。还支持 16 位的段寄存器（ES、CS、SS、DS、FS、GS），但是在 32 位代码中通常不使用段。

符号 ST 指示在 8087 浮点寄存器堆栈最顶端的寄存器，可以用 ST(X) 引用八位浮点寄存器中的每一个寄存器，X 是一个在 0~7 之间的常数，指示到寄存器顶端的距离。

(3) 符号

Delphi 的内嵌汇编器支持在汇编代码中存取几乎所有的 Object Pascal 标识符，包括常量、类型、变量、过程和函数。此外还支持特殊符号@Result，它对应的是函数内部的结果变量。下面的例子为函数 Sum 的形式：

```
function Sum(X, Y: Integer): Integer;
```

```
begin
```

```
    Result := X + Y;
```

```
end;
```

可以写成汇编代码如下：

```
function Sum(X, Y: Integer): Integer; stdcall;
```

```
begin
```

```
    asm
```

```
        MOV     EAX,X
```

```
        ADD     EAX,Y
```

```
        MOV     @Result,EAX
```

```
    end;
```

end;

下述的符号不能在汇编语句中使用:

- ❑ 标准过程和函数 (例如 WriteLn 和 Chr);
- ❑ Mem, MemW, MemL, Port 和 PortW 等特殊数组;
- ❑ 字符串变量, 浮点数和集合常量;
- ❑ 没在当前块声明的标签;
- ❑ 在函数外的 @Result 符号。

表 3.3 为在 asm 语句中可以使用的符号。

表 3.3 在 asm 语句中可使用的符号

符号	数值	类别	类型
label	标签地址	内存引用	SHORT
Const	常量数值	立即数	0
Type	0	内存引用	Size of type
Field	域偏移	内存引用	Size of type
Var	变量地址	内存引用	Size of type
Procedure	过程地址	内存引用	NEAR
Function	函数地址	内存引用	NEAR
Unit	0	立即数	0
@Code	代码段地址	内存引用	0FFF0H
@Data	数据段地址	内存引用	0FFF0H
@Result	结果变量偏移	内存引用	Size of type

由于编译优化在使用 asm 时无效, 局域变量总是被分配在堆栈中, 并且通过 EBP 存取, 局域变量的值来自由 EBP 指示的偏移, 汇编器在引用局域变量时将自动加上[EBP]内容。例如以下声明:

```
var
```

```
    Count: Integer;
```

asm 中的指令为:

```
MOV    EAX,Count
```

将自动汇编成 MOV EAX, [EBP-2]。

内嵌汇编器将 var 参数当作一个 32 位指针处理, 而且 var 参数的大小总是 4 (32 位指针的长度)。存取 var 参数与存取数值参数的语法是不同的, 要存取 var 参数的内容必须首先装载 32 位指针, 然后存取该指针指向的位置, 例如在前述 Sum 例子中, 将 X、Y 由常量改成变量, 则程序代码应当改为:

```
function Sum3(var X,Y :Integer): Integer;stdcall;
```

```
begin
```

```
asm
    MOV    EAX,X
    MOV    EAX,[EAX]
    MOV    EDX,Y
    ADD    EAX,[EDX]
    MOV    @Result, EAX
end;
end;
```

3. 表达式类别

在内嵌汇编程序中将表达式分为 3 种类别：寄存器、内存引用和立即数。

如果在表达式中只包含一个单独的寄存器，则称之为寄存器表达式，可以使用的寄存器如 AX、CL、DI 和 ES。由寄存器表达式生成的汇编指令将直接操作 CPU 寄存器。

指示内存地址的表达式为内存引用类别。Object Pascal 的标签、变量、类型常量、过程和函数属于这一类别。

如果表达式不是寄存器或不与内存地址关联，则是立即数类别，它包括 Object Pascal 的非类型常量、类型标识符。

使用立即数和内存引用时将会产生不同的程序代码，例如：

```
const
    Start = 10;
var
    Count: Integer;
...
asm
    MOV    EAX,Start           { MOV EAX,xxxx }
    MOV    EBX,Count          { MOV EBX,[xxxx] }
    MOV    ECX,[Start]        { MOV ECX,[xxxx] }
    MOV    EDX,OFFSET Count    { MOV EDX,xxxx }
end;
```

因为 Start 是个立即数，第一条 MOV 汇编成一条移动立即数指令，而第二条 MOV 汇编成一条移动内存指令，是因为 Count 是个内存引用。第三条 MOV 中，方括号运算符是用来将 Start 转换成一个内存引用（在例子中，word 在数据段的偏移是 10），在第四条 MOV 中，OFFSET 将 Count 转换为一个立即数（Count 的偏移在数据段中）。

方括号和 OFFSET 操作符相互补充。下面的 asm 语句给出了对应于上面 asm 语句中前两条语句的机器码：

```
asm
    MOV    EAX,OFFSET [Start]
    MOV    EBX,[OFFSET Count]
end;
```

4. 表达式类型

每个内嵌汇编表达式都有一个类型，更确切的是都具有长度，因为汇编器仅仅是简单地将表达式类型视作内存位置的长度。例如，Integer 型变量长度为 4，因为它占 4 个字节。内嵌汇编器在任何可能的情况下都将执行类型检查，因此对于下面的指令：

```
var
    QuitFlag: Boolean;
    OutBufPtr: Word;
...
asm
    MOV     AL,QuitFlag
    MOV     BX,OutBufPtr
end;
```

汇编器将检查 QuitFlag 的长度为 1（1 个字节），OutBufPtr 的长度为 2（一个字）。下面的指令执行时将会出错。

```
MOV     DL,OutBufPtr
```

产生错误的原因是因为 DL 是一个字节长度，OutBufPtr 是一个字。内存引用类型可以通过类型转换来改变类型，如下面 3 条指令：

```
MOV     DL,BYTE PTR OutBufPtr
MOV     DL,Byte(OutBufPtr)
MOV     DL,OutBufPtr.Byte
```

这些 MOV 指令都是引用 OutBufPtr 变量的第一个字节。

有些情况下，一个内存引用是无类型的，其中一个例子是方括号中的立即数：

```
MOV     AL,[100H]
MOV     BX,[100H]
```

内嵌汇编器支持这些指令，因为表达式[100H]没有连接类型，仅意味着“在数据段[100H]的内容”，类型是由一个操作数（AL 是字节，则 BX 是字）决定的。如果类型不能由其他的操作数决定，则需要一条类型转换语句，例如：

```
INC     BYTE PTR [100H]
IMUL    WORD PTR [100H]
```

表 3.4 列出了预定义的类型符号。

表 3.4 预定义的类型符号

符号	类型
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

5. 表达式运算符

内嵌汇编器提供了大量的运算符，其优先级顺序与 Object Pascal 有所不同，例如在 asm 语句中，AND 的优先级低于减 (-) 和加 (+) 运算符。

3.1.3 汇编程序过程和函数

Delphi 中可以用内嵌汇编代码编写完整的过程和函数，而不需要 begin...end 语句，例如：

```
function LongMul(X,Y:Integer):Longint;
```

```
asm
```

```
    MOV    EAX,X
```

```
    IMUL   Y
```

```
end;
```

编译器执行一系列的优化：

❑ 编译器不生成将数值参数复制到局部变量的程序代码，这影响所有字符串类型参数以及长度不是 1、2、4 字节的参数。在过程中这些参数必须处理成 var 参数。

❑ 除非函数返回一个字符串、变量或接口引用 (interface reference)，否则编译器不分配函数结果变量，引用 @Result 符号将出错。对于字符串、变量和界面元素总是为调用者分配 @Result 指针。

❑ 编译器不为无嵌套、无参数、无局部变量的过程和函数生成堆栈结构。

❑ 自动为汇编过程和函数生成程序入口和退出代码，如下所示：

```
PUSH    EBP
```

```
MOV     EBP,ESP
```

```
SUB     ESP,Locals
```

```
...
```

```
MOV     ESP,EBP
```

```
POP     EBP
```

```
RET     Params
```

如果 Locals 包含变量、长字符串或接口 (interface)，它们被初始化为 0，但没有最终确定下来。

Locals 是局部变量的大小，Params 是参数的大小。如果 Locals 和 Params 都为 0，则没有入口代码 (entry code)，退出代码 (exit code) 仅包含 RET 指令。

汇编函数将返回以下的结果：

❑ 序数类型值返回在 AL (8 位)、AX (16 位) 或 EAX (32 位)。

❑ 实数类型值返回在协处理器堆栈的 ST (0)。

❑ 指针包括长字符串返回在 EAX。

❑ 短字符串和变量返回在由指针 @Result 指向的临时位置。

3.2 嵌入式汇编的通信编程例子

下面给出几个 Delphi 嵌入式汇编的通信编程例子。

3.2.1 在 Delphi 中对端口的直接操作

Delphi 对端口的直接操作可以通过以下两种方式完成：

1. 预定义数组

Delphi 仍然保留了 Turbo Pascal 的两个预定义数组 Port 和 PortW。这两个一维数组的每个元素代表一个数据端口，它们的端口地址同它们的下标是相对应的，下标的类型为整型字。

Port 数组成员是 Byte 类型的，PortW 数组的成员是 Word 类型的。

下面举例说明：

```
Port[$20]:=$20;           //将$20 写入$20 端口
PortData:=Port[$20];      //将$20 端口的数据读入 Byte 变量 PortData
这种操作方式在 Delphi 的帮助文件中没有提及，但它确实可以应用。
```

2. 方法是采用直接嵌入式汇编语言

下面给出一个用于测试的例子：

```
Var
    PortData:Byte;
begin
    asm
        mov al,$20
        mov dx,$20
        out dx,al      //将$20 写入$20 端口
        mov dx,$20
        in  al,dx
        mov PortData,al; //将$20 端口的数据读入 Byte 变量 PortData
    end;
end;
```

3.2.2 行间汇编接收下位机传来的数据的简单例子

下面为 Delphi 中使用行间汇编接收下位机传来的数据的简单例子。该程序通过 COM2 接收 400 个传来的字符，并将这些内容保存在 aaa.dat 文件中，当接收完毕后，显示“Receive end”。但实际情况中双方通信可能约定的是传送字符的个数，也可能约定的是规定好的起始字符和结束字符，或是多种条件同时约定。待各种条件全部满足时才表示完成一次成功的接收；否则，只要有一个条件未满足就表示接收失败，需要重新传送。这些约定在使用了行间

汇编的 Delphi 程序中都可实现。下面的程序为按一个按钮进入的一个简单的串口接收程序。

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
    ca:array[1..400]of char;
```

```
    c:char;
```

```
    i,j:integer;
```

```
    f1:file of char;
```

```
    label loop1;
```

```
begin
```

```
    i:=1;
```

```
    asm
```

```
        mov dx,0001
```

```
        mov ax,005eh
```

```
        int 14h
```

```
    end;
```

```
    for j:=1 to 400 do
```

```
    begin
```

```
        asm
```

```
            loop1:
```

```
            mov dx,0001
```

```
            mov ah,02
```

```
            int 14h
```

```
            test ah,80h
```

```
            jnz loop1
```

```
            mov c,al
```

```
        end;    // 汇编结束 end of asm
```

```
        ca[i]:=c;
```

```
        i:=i+1;
```

```
    end;    // for j:=1 to 400 do 语句结束
```

```
    assignfile(f1,'aaa.dat');
```

```
    rewrite(f1);
```

```
    for j:=1 to i-1 do
```

```
        write(f1,ca[j]);
```

```
    closefile(f1);
```

```
    label1.caption:= 'receive end';
```

```
end;
```

3.2.3 用于串行通信的 Delphi DLL 程序

下面例子为一个用于串行通信的 Delphi DLL 程序。

```

library Mycomdll;
uses
    SysUtils,Classes;
const
    COM1=$3f8;//定义串口字符常量  COM1 的 I/O 地址为 0x3f8
    COM2=$2f8;                      //COM2 的 I/O 地址为 0x2f8
var
    COM:Word;//DLL 中全局变量
procedure Outb(Const Port:Word;Const Dbyte:Byte);pascal;
begin
    asm
        MOV DX,Port
        MOV AL,Dbyte
        OUT DX,AL
    end;
end;

function Inb(Const Port:Word):Byte;pascal;
begin
    asm
        MOV DX,Port
        IN  AL,DX
        MOV @Result,AL
    end;
end;

//串行口初始化, COM1、COM2 可选, 波特率 2400、4800 可选
procedure CommInit(CONST Port:Byte;CONST Baud:String);Stdcall;
var
    BAUDL,BAUDH:Byte;
begin
    if Port=1 then
        COM:=COM1
    else
        COM:=COM2;
    if BAUD='2400' then
        begin
            BAUDL:=$30;
            BAUDH:=$00;
        end;
    end;
end;

```



```
    end
    else
    begin
        BAUDL:=$18;
        BAUDH:=$00;
    end;
    outb(COM+3,$80); //设置波特率因子
    outb(COM,BAUDL);
    outb(COM+1,BAUDH);
    outb(COM+3,$03); //8 位数据, 1 停止位, 无校验
end;

//发送 1Byte
procedure SendByte(Const Dbyte:Byte);Stdcall;
var
    status:Byte;
begin
    repeat
        status:=Inb(COM+5);
    until((Status and $20)=$20);
    outb(COM,Dbyte);
end;

//接收 1Byte
function ReceiveByte:Byte;Stdcall;
var
    status,res:Byte;
begin
    repeat
        status:=Inb(COM+5);
    until((Status and $01)=$01);
    res:=Inb(COM);
    receiveByte:=Res;
end;

//以下用 exports 引出输出过程或函数
exports CommInit;
exports SendByte;
exports ReceiveByte;
begin
end.
```

3.2.4 直接操作端口的 Delphi 单元

用 Delphi 来编写工业控制程序时，需要对计算机所连接的外部设备进行操作，即直接对 I/O 地址进行读写操作，这时纯粹的 Pascal 语言就显得有些美中不足了。针对这一问题，笔者使用 Delphi 以内嵌汇编的方式编写了一个模块 Port.pas，可方便地实现直接对 I/O 地址的读写操作，代码简捷且执行速度较快。

使用时只要将 Port.pas 加到工程文件中，并在 uses 中加上 Port，就可以在应用程序中直接对 I/O 端口进行操作。

具体的实现方法及 Port.pas 的源代码如下：

```
unit Port;
interface

function PortReadByte(Addr:Word) : Byte;
function PortReadWord(Addr:Word) : Word;
function PortReadWordLS(Addr:Word) : Word;
procedure PortWriteByte(Addr:Word; Value:Byte);
procedure PortWriteWord(Addr:Word; Value:Word);
procedure PortWriteWordLS(Addr:Word; Value:Word);

implementation
    //PortReadbyte 函数
    //参数: port address
    //返回: 给定 port 的 byte 值
function PortReadByte(Addr:Word) : Byte; assembler; register;
asm
    MOV DX,AX
    IN AL,DX
end;
    //高速读端口函数: PortReadWord 函数
    //参数: port address
    //返回: 给定 port 的 word 值
    //注释: 可能有些卡和计算机不能访问全部的 word
function PortReadWord(Addr:Word) : Word; assembler; register;

asm
    MOV DX,AX
    IN AX,DX
end;
    //低速读端口函数
```

//参数: port address

//返回: 给定 port 的 word 值

//注释: 工作时, 要调整 DELAY

function PortReadWordLS(Addr: Word): Word; assembler; register;

const

Delay = 150;

//依靠 CPU 的速度和卡的速度

asm

MOV DX,AX

IN AL,DX

//读 LSB 端口

MOV ECX,Delay

@1:

LOOP @1 //在两次读之间延时

XCHG AH,AL

INC DX

//port+1

IN AL,DX //读 MSB 端口 read MSB port

XCHG AH,AL //重新存储字节顺序

end;

//PortWriteByte 函数

procedure PortWriteByte(Addr: Word; Value: Byte); assembler; register;

asm

XCHG AX,DX

OUT DX,AL

end;

//高速写端口过程

//注释: 工作时, 可能有些卡和计算机不能访问全部的 word

procedure PortWriteWord(Addr: word; Value: word); assembler; register;

asm

XCHG AX,DX

OUT DX,AX

end;

//低速写端口过程

procedure PortWriteWordLS(Addr: word; Value: word); assembler; register;

const

Delay = 150;

//依靠 CPU 的速度和卡的速度

```
asm
    XCHG AX,DX
    OUT DX,AL
    MOV ECX,Delay
    @1:
    LOOP    @1
    XCHG AH,AL
    INC DX
    OUT DX,AL
end;

end. //单元结束
```

本章小结

本章首先介绍 Delphi 中的嵌入式汇编语言的语法和指令、表达式元素、汇编程序过程和函数，然后给出了直接操作端口的 Delphi 单元、用于串行通信的 Delphi DLL 程序等内嵌汇编语言的通信编程例子。

第4章 MSComm 控件应用

本章主要内容:

- ☐ MSComm 控件的方法
- ☐ MSComm 控件的属性
- ☐ MSComm 控件的事件
- ☐ MSComm 程序实例和分析

Microsoft Communication Control (以下简称为 MSComm) 是 Microsoft 公司提供的 Windows 下串行通信编程的 ActiveX 控件。MSComm 控件是 Visual Basic 中的 OCX 控件。VB 的 MSComm 通信控件具有丰富的与串口通信密切相关的属性及事件, 提供了一系列标准通信命令的接口, 可以用它创建全双工的、事件驱动的、高效实用的通信程序。

4.1 MSComm 控件

在 Delphi 3.0 中无法使用 MSComm 控件, 笔者使用的是 Delphi 5.0。在 Delphi 中安装 MSComm 控件, 先打开 Delphi 5.0 集成开发环境, 选择菜单“Component”中的“Import ActiveX Control”命令, 在“Import ActiveX”选项卡内选择“Microsoft Comm Control 6.0”项, 如图 4.1 所示。

单击“Install”按钮安装 MSComm 控件, 安装后在“ActiveX”组件板中出现 MSComm 图标, 即可被使用。有一点要注意, 在 Object Inspector 中 MSComm 控件的 Input 和 Output 属性是不可见的, 但它们仍然存在, 这两个属性的类型是 OleVariant (Ole 万能变量)。

4.1.1 MSComm 控件方法

MSComm 控件通过串行端口传输和接收数据, 为应用程序提供串行通信功能。

MSComm 控件提供下列两种处理通信的方式:

(1) 事件驱动通信是处理串行端口交互作用的一种非常有效的方法。在许多情况下, 在事件发生时需要得到通知, 例如, 在 Carrier Detect (CD) 或 Request To Send (RTS) 信号线上一个字符到达或一个变化发生时。在这些情况下, 可以利用 MSComm 控件的 OnComm 事件捕获并处理这些通信事件。OnComm 事件还可以检查和处理通信错误。

(2) 在程序的每个关键功能之后, 可以通过检查 CommEvent 属性的值来查询事件和错误。如果应用程序较小, 并且是自保持的, 这种方法可能是更可取的。例如, 如果写一个简单的电话拨号程序, 则没有必要对每接收一个字符都产生事件, 因为唯一等待接收的字符是

调制解调器的“确定”响应。



图 4.1 Import ActiveX

每个 MSComm 控件对应着一个串行端口。如果应用程序需要访问多个串行端口，必须使用多个 MSComm 控件。可以在 Windows “控制面板”中改变端口地址和中断地址。

尽管 MSComm 控件有很多重要的属性，但首先必须熟悉几个属性。

CommPort	设置并返回通信端口号。
Settings	以字符串的形式设置并返回波特率、奇偶校验、数据位、停止位。
PortOpen	设置并返回通信端口的状态，也可以打开和关闭端口。
Input	从接收缓冲区返回和删除字符。
Output	向传输缓冲区写一个字符串。

4.1.2 MSComm 控件属性

通信 MSComm 控件提供了 27 个关于通信控制方面的属性和 5 个标准属性，Delphi 5.0 中 MSComm 控件的属性图如图 4.2 所示。

下面介绍它的主要属性：

□ Break 属性

描述：设置或清除中断信号的状态。该属性在设计时无效。

语法：

Visual Basic

[form.]MSComm.Break[={ True |False}]

Delphi

[form.]MSComm.Break[:={ True |False}]

设置为：

True	设置中断信号状态
False	清除中断信号状态

注释：当设置为 True，Break 属性发送一个中断信号。该中断信号挂起字符传输，并置传输线为中断状态直到把 Break 属性设置为 False。一般，仅当使用的通信设备要求设置一个中断信号时，才设置一个短时的中断状态。

数据类型：Integer(Boolean)



图 4.2 MSComm 控件的属性图

□ CDHolding 属性

通过查询 Carrier Detect (CD) 信号线的状态确定当前是否有传输。Carrier Detect 是从调制解调器发送到相连计算机的一个信号，指示调制解调器正在联机。该属性在设计时无效，在运行时为只读。

语法：

Visual Basic

[form.]MSComm.CDHolding[={True|False}]

Delphi

[form.]MSComm.CDHolding[:={True|False}]

CDHolding 属性的设置值为：

True Carrier Detect 信号线为高电平

False Carrier Detect 信号线为低电平

注释：当 Carrier Detect 信号线为高电平 (CDHolding=True) 且超时，MSComm 控件设置 CommEvent 属性为 ComEventCDTO (Carrier Detect 超时错误)，并产生 OnComm 事件。

□ CommID 属性

返回一个说明通信设备的句柄。该属性在设计时无效，在运行时为只读。

语法：

Visual Basic

[form.]MSComm.CommID

Delphi

[form.]MSComm.CommID

注释：该值与 Windows API CreateFile 函数返回的值一致。在 Windows API 中调用任何通信例程时使用该值。

数据类型：Long

□ CommEvent 属性

返回最近的通信事件或错误。该属性在设计时无效，在运行时为只读。

语法：

Visual Basic

[form.]MSComm.CommEvent

Delphi

[form.]MSComm.CommEvent

注释：只要有通信错误或事件发生时都会产生 OnComm 事件，CommEvent 属性存有该错误或事件的数值代码。要确定引发 OnComm 事件的确切的错误或事件，请参阅 CommEvent 属性。

数据类型：Integer

CommEvent 属性返回下列值之一来表示不同的通信错误或事件。这些常数可以在该控件的对象库中找到。通信错误常数见表 4.1。

表 4.1 通信错误常数

常数	值	含义描述
ComEventBreak	1001	接收到一个中断信号
ComEventCTSTO	1002	Clear To Send 超时。在系统规定时间内传输一个字符时，Clear To Send 信号线为低电平
ComEventDSRTO	1003	Data Set Ready 超时。在系统规定时间内传输一个字符时，Data Set Ready 信号线为低电平
ComEventFrame	1004	帧错误。硬件检测到一帧错误
comEventOverrun	1006	端口超速。没有在下一个字符到达之前从硬件读取字符，该字符丢失
comEventCDTO	1007	载波检测超时。在系统规定时间内传输一个字符时，Carrier Detect 信号线为低电平。Carrier Detect 也称为 Receive Line Signal Detect(RLSD)
comEventRxOver	1008	接受缓冲区溢出。接收缓冲区没有空间
comEventRxParity	1009	奇偶校验。硬件检测到奇偶校验错误
comEventTxFull	1010	传输缓冲区已满。传输字符时传输缓冲区已满
comEventDCB	1011	检索端口的设备控制块 (DCB) 时的意外错误

通信事件常数如表 4.2 所示。

表 4.2 通信事件常数

常数	值	含义描述
comEvSend	1	在传输缓冲区中有比 Sthreshold 数少的字符
comEvReceive	2	收到 Rthreshold 个字符。该事件将持续产生直到用 Input 属性从接收缓冲区中删除数据
comEvCTS	3	Clear To Send 信号线的状态发生变化
comEvDSR	4	Data Set Ready 信号线的状态发生变化。该事件只在 DST 从 1 变到 0 时才发生
comEvCD	5	Carrier Detect 信号线的状态发生变化
ComEvRing	6	检测到振铃信号。一些 UART (通用异步接收—传输) 可能不支持该事件
ComEvEOF	7	收到文件结束 (ASCII 字符为 26) 字符

□ CommPort 属性

设置并返回通信端口号。

语法:

Visual Basic

[form.]MSComm.CommPort[=value]

Delphi

[form.]MSComm.CommPort[:=value]

注释:在设计时,value 可以设置成从 1 到 16 的任何数(缺省值为 1)。但是如果用 PortOpen 属性打开一个并不存在的端口时,MSComm 控件会产生错误 68 (设备无效)。

注意:必须在打开端口之前设置 CommPort 属性。

数据类型: Integer

□ CTSHolding 属性

确定是否可通过查询 Clear To Send (CTS) 信号线的状态发送数据。Clear To Send 是调制解调器发送到相连计算机的信号,指示传输可以进行。该属性在设计时刻不能设置,在运行时刻只能读不能写。

语法:

Visual Basic

[form.]MSComm.CTSHolding[=(True|False)]

Delphi

[form.]MSComm.CTSHolding[:=(True|False)]

CTSHolding 属性设置值:

True Clear To Send 信号线为高电平

False Clear To Send 信号线为低电平

注释:如果 Clear To Send 信号线为低电平 (CTSHolding=False) 并且超时,MSComm 控件设置 CommEvent 属性为 ComEventCTSTO (Clear To Send Timeout) 并产生 OnComm 事

件。

Clear To Send 信号线用于 RTS/CTS (Request To Send/Clear To Send) 硬件握手。如果需要确定 Clear To Send 信号线的状态, CTS Holding 属性给出一种手工查询的方法。

数据类型: Boolean

❑ DSRHolding 属性

确定 Data Set Ready (DSR) 信号线的状态。Data Set Ready 信号由调制解调器发送到相连计算机, 指示作好操作准备。该属性在设计时无效, 在运行时为只读。

语法:

Visual Basic

```
[form.]MSComm.DSRHolding[={True|False}]
```

Delphi

```
[form.]MSComm.DSRHolding[:={True|False}]
```

DSRHolding 属性返回以下值:

True Data Set Ready 信号线为高电平

False Data Set Ready 信号线为低电平

注释: 当 Data Set Ready 信号线为高电平 (DSRHolding=True) 且超时, MSComm 控件设置 CommEvent 属性为 comEventDSRTO (数据准备超时) 并产生 OnComm 事件。当为 Data Terminal Equipment (DTE) 机器写 Data Set Ready/Data Terminal Ready 握手例程时该属性是十分有用的。

数据类型: Boolean

❑ DTREnable 属性

确定在通信时是否使 Data Terminal Ready (DTR) 信号线有效。Data Terminal Ready 是计算机发送到调制解调器的信号, 指示计算机在等待接受传输。

语法:

Visual Basic

```
[form.]MSComm.DTREnable[={True|False}]
```

Delphi

```
[form.]MSComm.DTREnable[:={True|False}]
```

DTREnable 属性设置值:

True 使 Data Terminal Ready 信号线有效

False 使 Data Terminal Ready 信号线无效 (缺省)

注释: 当 DTREnable 设置为 True, 当端口被打开时 Data Terminal Ready 信号线设置为高电平 (开), 当端口被关闭时 Data Terminal Ready 信号线设置为低电平 (关)。当 DTREnable 设置为 False, Data Terminal Ready 信号线始终保持为低电平。

注意: 在很多情况下, 把 Data Terminal Ready 信号线设置为低电平用来挂断电话。

数据类型: Boolean

范例: 下例表示数据终端准备好。

```
MSComm1.DTREnable:=True;
```

❑ EOFEnable 属性

EOFEnable 属性确定在输入过程中 MSComm 控件是否寻找文件结尾 (EOF) 字符。如果找到 EOF 字符, 将停止输入并激活 OnComm 事件, 此时 CommEvent 属性设置为 ComEvEOF。

语法:

Visual Basic

[form.]MSComm.EOFEnable[={True|False}]

Delphi

[form.]MSComm.EOFEnable[:={True|False}]

value 的设置值:

True 当 EOF 字符找到时 OnComm 事件被激活。

False 当 EOF 字符找到时 OnComm 事件不被激活 (缺省)。

注释: 当 EOFEnable 属性设置为 False, OnComm 控件将不在输入流中寻找 EOF 字符。

□ Handshaking 属性

设置并返回硬件握手协议。

语法:

Visual Basic

[form.]MSComm.Handshaking[=value]

Delphi

[form.]MSComm.Handshaking[:=value]

value 设置值见表 4.3。

注释: Handshaking 是指内部通信协议, 通过该协议, 数据从硬件端口传输到接收缓冲区。当一个数据字符到达串行端口, 通信设备就把它移到接收缓冲区以使程序可以读它。如果没有接收缓冲区, 程序需要直接从硬件读取每一个字符, 这很可能会造成数据丢失, 因为字符到达的速度可以非常快。

握手协议保证在缓冲区过载时数据不会丢失。缓冲区过载为数据到达端口太快而使通信设备来不及将它移到接收缓冲区。

数据类型: Integer

表 4.3

Handshaking 属性的 value 设置值

常数	值	含义描述
comNone	0	没有握手 (缺省)
comXOnXOff	1	(XON/XOFF) 握手
comRTS	2	RTS/CTS (Request To Send/Clear To Send) 握手
comRTSXOnXOff	3	Request To Send 和 XON/XOFF 握手皆可

□ InBufferCount 属性

返回接收缓冲区中等待的字符数。该属性在设计时无效。

语法:

Visual Basic

[form.]MSComm.InBufferCount[=value]

Delphi

[form.]MSComm.InBufferCount[:=value]

注释: InBufferCount 是指调制解调器已接收, 并在接收缓冲区等待被取走的字符数。可以把 InBufferCount 属性设置为 0 来清除接收缓冲区。

注意: 不要把该属性与 InBufferSize 属性混淆。InBufferSize 属性返回整个接收缓冲区的大小。

数据类型: Integer

□ InBufferSize 属性

设置并返回接收缓冲区的字节数。

语法:

Visual Basic

[form.]MSComm.InBufferCount[=value]

Delphi

[form.]MSComm.InBufferCount[:=value]

注释: InBufferSize 是指整个接收缓冲区的大小。缺省值是 1024 字节。不要将该属性与 InBufferCount 属性混淆, InBufferCount 属性返回的是当前在接收缓冲区中等待的字符数。注意接收缓冲区越大则应用程序可用内存越小。但若接收缓冲区太小, 且不使用握手协议, 就可能有溢出的危险。一般的规律是, 首先设置一个 1024 字节的缓冲区, 如果出现溢出错误, 则通过增加缓冲区的大小来控制应用程序的传输速率。

数据类型: Integer

□ Input 属性

返回并删除接收缓冲区中的数据流。该属性在设计时无效, 在运行时为只读。

语法:

Visual Basic

[form.]MSComm.Input

Delphi

[form.]MSComm.Input

注释: InputLen 属性确定被 Input 属性读取的字符数。设置 InputLen 为 0, 则 Input 属性读取缓冲区中全部的内容。InputMode 属性确定用 Input 属性读取的数据类型。如果设置 InputMode 为 comInputModeText, Input 属性通过一个 Variant 返回文本数据。如果设置 InputMode 为 comInputModeBinary, Input 属性通过一个 Variant 返回一二进制数据的数组。

数据类型: Variant

□ InputLen 属性

设置并返回 Input 属性从接收缓冲区读取的字符数。

语法:

Visual Basic

[form.]MSComm.InputLen[=value]

Delphi

[form.]MSComm.InputLen[:=value]

注释: InputLen 属性的缺省值是 0。设置 InputLen 为 0 时, 使用 Input 将使 MSComm 控件读取接收缓冲区中全部的内容。

若接收缓冲区中 InputLen 字符无效, Input 属性返回一个零长度字符串("")。在使用 Input 前, 用户可以选择检查 InBufferCount 属性来确定缓冲区中是否已有所需数目的字符。

该属性在从输出格式为定长数据的计算机读取数据时非常有用。

数据类型: Integer

下例说明如何读取 10 个字符数据。

```
MSComm1.InputLen:=10;
```

```
//指定 10 个字符的数据块
```

```
CommData:=MSComm1.Input;
```

```
//读取数据
```

□ InputMode 属性

设置或返回 Input 属性取回的数据的类型。

语法:

Visual Basic

[form.]MSComm.InputMode[=value]

Delphi

[form.]MSComm.InputMode[:=value]

value 设置值如表 4.4 所示。

表 4.4 InputMode 属性的 value 设置值

常数	值	含义描述
ComInputModeText	0	数据通过 Input 属性以文本形式取回 (缺省)
ComInputModeBinary	1	数据通过 Input 属性以二进制形式取回

注释: InputMode 属性确定 Input 属性如何取回数据。数据取回的格式或是字符串或是二进制数据的数组。若数据只用 ANSI 字符集, 则用 comInputModeText。对于其他字符数据, 如数据中有嵌入控制字符、Nulls 等等, 则使用 comInputModeBinary。

□ NullDiscard 属性

确定 NULL 字符是否从端口传送到接收缓冲区。

语法:

Visual Basic

[form.]MSComm.NullDiscard[=value]

Delphi

[form.]MSComm.NullDiscard[:=value]

value 设置值是:

True NULL 字符不从端口传送到接收缓冲区

False NULL 字符从端口传送到接收缓冲区 (缺省值)

注释: NULL 字符定义为 ASCII 字符 0。

数据类型: Boolean

❑ OutBufferCount 属性

返回在传输缓冲区中等待的字符数, 也可以用它来清除传输缓冲区。该属性在设计时无效。

语法:

Visual Basic

[form.]MSComm.OutBufferCount[=value]

Delphi

[form.]MSComm.OutBufferCount[:=value]

注释: 设置 OutBufferCount 属性为 0 可以清除传输缓冲区。注意不要把 OutBufferCount 属性与 OutBufferSize 属性混淆, OutBufferSize 属性返回整个传输缓冲区的大小。

数据类型: Integer

下例表示清空输出缓冲区。

```
MSComm1.OutBufferCount:=0;
```

❑ OutBufferSize 属性

以字节的形式设置并返回传输缓冲区的大小。

语法:

Visual Basic

[form.]MSComm.OutBufferSize[=value]

Delphi

[form.]MSComm.OutBufferSize[:=value]

注释: OutBufferSize 指整个传输缓冲区的大小, 缺省值是 512 字节。不要把该属性与 OutBufferCount 属性混淆, OutBufferCount 属性返回当前在传输缓冲区等待的字节数。

注意: 传输缓冲区设置得越大则应用程序可用内存越小。但若缓冲区太小, 若不使用握手协议, 就可能有溢出的危险。一般的规律是, 首先设置一个 512 字节的缓冲区。如果出现溢出错误, 则通过增加缓冲区的大小来控制应用程序的传输速率。

数据类型: Integer

❑ Output 属性

往传输缓冲区写数据流。该属性在设计时无效, 在运行时为只读。

语法:

Visual Basic

[form.]MSComm.Output[=value]

Delphi

[form.]MSComm.Output[:=value]

注释: Output 属性可以传输文本数据或二进制数据。用 Output 属性传输文本数据, 必须定义一个包含一个字符串的 Variant。发送二进制数据, 必须传递一个包含字节数组的 Variant 到 Output 属性。正常情况下, 如果发送一个 ANSI 字符串到应用程序, 可以以文本数据的形式发送。如果发送包含嵌入控制字符、Null 字符等等的的数据, 要以二进制形式发送。

数据类型: Variant

下例表示传输 AT 命令数据。

```
MSComm1.Output:='ATZ'+chr(13);
```

☐ ParityReplace 属性

当发生奇偶校验错误时, 设置并返回替换数据流中一个非法字符的字符。

语法:

Visual Basic

```
[form.]MSComm.ParityReplace[=value]
```

Delphi

```
[form.]MSComm.ParityReplace[:=value]
```

注释: Parity bit 是指同一定数据位数一起传输的位, 以提供简单的错误检查。当使用校验位时, MSComm 控件把在数据中已经设置的所有位 (值为 1) 都加起来并检查其和为奇数或偶数 (根据当端口开时奇偶校验的设置)。

按照缺省规定, MSComm 控件用问号 (?) 替换非法字符。若设置 ParityReplace 为一个空字符串 (""), 则当奇偶校验错误出现时, 字符替换无效。

数据类型: String

☐ PortOpen 属性

设置并返回通信端口的状态 (开或关)。该属性在设计时无效, 在运行时刻才可用。

语法:

Visual Basic

```
[form.]MSComm.PortOpen[=value]
```

Delphi

```
[form.]MSComm.PortOpen[:=value]
```

value 设置值是:

True 端口开

False 端口关

注释: 设置 PortOpen 属性为 True 打开端口。设置为 False 关闭端口并清除接收和传输缓冲区。当应用程序终止时, MSComm 控件自动关闭串行端口。

在打开端口之前, 确定 CommPort 属性设置为一个合法的端口号。如果 CommPort 属性设置为一个非法的端口号, 则当打开该端口时, MSComm 控件产生错误 68 (设备无效)。另外, 串行端口设备必须支持 Settings 属性当前的设置值。如果 Settings 属性包含硬件不支持的通信设置值, 那么硬件可能不会正常工作。如果在端口打开之前, DTREnable 或 RTSEnable 属性设置为 True, 当关闭端口时, 该属性设置为 False。否则, DTR 和 RTS 信号线保持其先前的状态。

数据类型: Boolean

☐ RThreshold 属性

在 MSComm 控件设置 CommEvent 属性为 comEvReceive 并产生 OnComm 之前, 设置并返回要接收的字符数。

语法:

Visual Basic

[form.]MSComm.Rthreshold[=value]

Delphi

[form.]MSComm.Rthreshold[:=value]

注释: 当接收字符后, 若 Rthreshold 属性设置为 0 (缺省值) 则不产生 OnComm 事件。例如, 设置 Rthreshold 为 1, 接收缓冲区收到每一个字符都会使 MSComm 控件产生 OnComm 事件。

数据类型: Integer

□ RTSEnable 属性

确定是否使 Request To Send (RTS) 信号线有效。一般情况下, 由计算机发送 Request To Send 信号到联接的调制解调器, 以请示允许发送数据。

语法:

Visual Basic

[form.]MSComm.RTSEnable[=value]

Delphi

[form.]MSComm.RTSEnable[:=value]

value 设置值:

True Request To Send 信号线有效

False Request To Send 信号线无效 (缺省)

注释: 当 RTSEnable 设置为 True, 端口打开时, Request To Send 信号线设置为高电平, 端口关闭时, 设置为低电平。Request To Send 信号线用在 RTS/CTS 硬件握手。RTSEnable 属性允许手动检测 Request To Send 信号线以确定其状态。

数据类型: Boolean

□ Settings 属性

设置并返回波特率、奇偶校验、数据位和停止位参数。

Visual Basic

[form.]MSComm.Settings[=value]

Delphi

[form.]MSComm.Settings[:=value]

注释: 当端口打开时, 如果 value 非法, 则 MSComm 控件产生错误 380 (非法属性值)。Value 由 4 个设置值组成, 有如下的格式:

"BBBB,P,D,S"

其中 BBBB 为波特率, P 为奇偶校验, D 为数据位数, S 为停止位数。value 的缺省值是:

"9600,N,8,1"

合法的波特率有:

110

300

600

1200

2400

9600 (缺省)

14000

19200

38400 (保留)

56000 (保留)

128000 (保留)

256000 (保留)

合法的奇偶校验值有:

E 偶数 (Even)

M 标记 (Mark)

N 缺省 (Default)

O 奇数 (Odd)

S 空格 (Space)

合法的数据位值有:

4

5

6

7

8 (缺省)

合法的停止位值有:

1 (缺省)

1.5

2

数据类型: String

下例表示将通信口设置为 56000bit/s、无奇偶校验、8 个数据位、1 个停止位。

```
MSComm1.Setting:='56000,N,8,1';
```

❑ SThreshold 属性

在 MSComm 控件设置 CommEvent 属性为 comEvSend 并产生 OnComm 事件之前, 设置并返回传输缓冲区中允许的最小字符数。

语法:

Visual Basic

```
[form.]MSComm.SThreshold[=value]
```

Delphi

```
[form.]MSComm.SThreshold[:=value]
```

数据类型: Integer

注释: 若设置 Sthreshold 属性为 0 (缺省值), 数据传输事件不会产生 OnComm 事件。若设置 Sthreshold 属性为 1, 当传输缓冲区完全空时, MSComm 控件产生 OnComm 事件。如果

在传输缓冲区中的字符数小于 value, CommEvent 属性设置为 comEvSend, 并产生 OnComm 事件。comEvSend 事件仅当字符数与 Sthreshold 交叉时被激活一次。例如, 如果 Sthreshold 等于 5, 仅当在输出队列中字符数从 5 降到 4 时, comEvSend 才发生。如果在输出队列中从没有比 Sthreshold 多的字符, comEvSend 事件将绝不会发生。

4.1.3 MSComm 控件事件的介绍

□ OnComm 事件

无论何时当 CommEvent 属性的值变化时, 就产生 OnComm 事件, 标志发生了一个通信事件或一个错误。

语法:

Visual Basic

```
Sub MSComm_OnComm()
```

Delphi

```
procedure MSCommComm(Sender:TObject);
```

注释: CommEvent 属性包含实际错误或产生 OnComm 事件的数字。注意, 设置 Rthreshold 或 Sthreshold 属性为 0, 分别使捕获 comEvReceive 和 comEvSend 事件无效。

以下的代码演示了如何控制通信的错误和事件, 读者可以插入代码到相关联的 CASE 语句中来处理特定的错误和事件。

Case MSComm1.CommEvent of

```
ComEventBreak: statement;
comEventCDTO: statement;
comEventCTSTO: statement;
comEventDSRTO: statement;
comEventFrame: statement;
comEventOverrun: statement;
comEventRxOver: statement;
comEventRxParity: statement;
comEventTxFull: statement;
comEventDCB: statement;
```

```
comEvCD: statement;
comEvCTS: statement;
comEvDSR: statement;
comEvRing: statement;
comEvReceive: statement;
comEvSend: statement;
comEvEOF: statement17;
```

//COM1

```
//Break 产生, 插入处理代码
//CD (RLSD) 超时, 插入处理代码
//CTS 超时, 插入处理代码
//DSR 超时, 插入处理代码
//Framing 错, 插入处理代码
//数据丢失, 插入处理代码
//接收缓冲溢出, 插入处理代码
//奇偶校验错, 插入处理代码
//发送缓冲区满, 插入处理代码
//异常产生, 插入处理代码
//通信事件
//CD 信号线改变, 插入处理代码
//CTS 信号线改变, 插入处理代码
//DSR 信号线改变, 插入处理代码
//Ring 指示器改变, 插入处理代码
//接收字符数小于阈值, 插入处理代码
//发送字符数小于阈值, 插入处理代码
//到达文件末尾, 插入处理代码
```

4.2 MSComm 控件的错误消息

MSComm 控件的错误 (Error) 常数见表 4.5。

表 4.5 MSComm 控件的错误 (Error) 常数

常数	值	含义描述
comEventBreak	1001	接收到中断信号
comEventCTSTO	1002	Clear-to-send 超时
comEventDSRTO	1003	Data-set ready 超时
comEventFrame	1004	帧错误
comEventOverrun	1006	端口超速
comEventCDTO	1007	Carrier Detect 超时
comEventRxOver	1008	接收缓冲区溢出
ComEventRxParity	1009	Parity 错误
ComEventTxFull	1010	传输缓冲区满
ComEventDCB	1011	检索端口的设备控制块时出现的意外错误

4.3 用 MSComm 控件编程实例

下面介绍简单的 MSComm 程序分析和复杂的 MSComm 程序实例和分析。

4.3.1 简单的 MSComm 程序分析

1. 例 1

在 Form 中放置一个 Memo 控件用于显示接收的数据, Combobox1 选择通信参数 (Setting 属性值), Combobox2 选择串口 (CommPort 属性值), 按 Button1 开始接收数据, 按 Button2 停止接收。窗体 Form 如图 4.3 所示。



图 4.3 窗体 Form 图

部分源代码如下:

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    MSComm1.InBufferCount :=0;      //清空接收缓冲区
    MSComm1.InputLen :=0;           //Input 读取整个缓冲区内容
    MSComm1.RThreshold:=1;          //每次接收到字符即产生 OnComm 事件
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    MSComm1.Settings :=ComboBox1.Text;
    if ComboBox2.Text ='com1' then  //假设只考虑 com1 和 com2 两种情况
        MSComm1.CommPort :=1
    else
        MSComm1.CommPort :=2;
    MSComm1.PortOpen :=true;        //打开串口
    MSComm1.DTREnable :=true;       //数据终端准备好
    MSComm1.RTSEnable :=true;       //请求发送
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    MSComm1.PortOpen :=false;       //关闭串口
    MSComm1.DTREnable :=false;
    MSComm1.RTSEnable :=false;
end;

procedure TForm1.MSComm1Comm(Sender: TObject);
var
    recstr:Olevariant;
begin
    if MSComm1.CommEvent = 2 then
    begin
        recstr := MSComm1.Input ;
        Memo1.text := Memo1.Text + recstr;
    end;
end;

```

2. 例 2

下面再结合一个具体的实例来说明如何用 MSComm 控件开发出串口通信程序。

创建一个 Communication.dpr 工程，把窗体的 Name 属性变为 CommForm，将标题改为 The Communication Test，选择 File/Save As 将新的窗体存储为 CommFrm.pas，接下来参照图 4.4 将图中的控件添加到主窗体中。窗体 CommForm 如图 4.4 所示。

由图 4.4 可以看出，通过设置页可选定进行数据传输的通信端口和端口的波特率、奇偶校验、数据位和停止位，通信时每传输一个字符都将触发响应事件，在通信页“传输显示”位置可看到当前正在进行传输的数据。当出现回车换行符时整行内容将显示在 memDisplay 新的一行中，而全部接收的内容还将存在一个文件中。



图 4.4 窗体 CommForm 图

部分源代码如下：

```
var
    CommForm: TCommForm;
    ss:string;
    savef,readf :file of char;
    i,j :longint;
//初始化
procedure TCommForm.FormCreate(Sender: TObject);
begin
    MSComm.commport:=1;
    MSComm.settings:='5600,n,8,1';
    //将通信口设置为 56000bit/s、无奇偶校验、8 个数据位、1 个停止位
    MSComm.inputlen:=1;
    MSComm.inbuffercount:=0;
    MSComm.portopen:=true;
    ss:="";
    i:=0;
    j:=0;
    assignfile(savef, SaveDialog1.FileName);
    rewrite(savef);
```

```

        assignfile(readf, OpenFileDialog1.FileName);
        reset(readf);
    end;

//设置确定
procedure TCommForm.btnConfirmClick(Sender: TObject);
begin
    if MSComm.portopen then
        MSComm.portopen:=false;
        MSComm.commport:=StrToInt(edtCommport.text);
        MSComm.settings:=edtCommsetting.Text;
    end;

//通信时每传输一个字符都将触发响应事件，在通信页“传输显示”位置可看到当前正
//在进行传输的数据
procedure TCommForm.MSCommComm(Sender: TObject);
var
    filenrc :char;
    buffer :variant;
    sl:string;
    c :char;
begin
    case MSComm.commEvent of
    comEvSend:
    begin
        while not(eof(readf)) do
        begin
            read(readf,filenrc);
            MSComm.output:=filenrc;
            j:=j+1;
            lblDisplay.caption:=inttostr(j);
            if MSComm.outbuffercount>=2 then
                break;
            end;
        end;
    end;
    comEvReceive:
    begin
        buffer:=MSComm.Input;
        sl:=buffer;
    end;
end;

```

```

c:=sl[1];
ss:=ss+c;
i:=i+1;
lblDisplay.caption:=c+inttostr(i);
write(savef,c);
//当出现回车换行符时将整行内容将显示在 memDisplay 新的一行中，而全部接收
//的内容还将存在一个文件中
if (c=chr(10))or(c=chr(13)) then
begin
    lblDisplay.caption:='cr'+IntToStr(i);
    memDisplay.lines.add(ss);
    ss:="";
end;
end;
end;
end;

```

4.3.2 复杂的 MSComm 程序实例和分析

下面给出一个控制 Modem 的基本库单元 cModem。该单元是笔者为一个项目编写的，它详细地实现了一个类 TModem，其中对 Modem 的控制是通过 MSComm 控件来实现的。

1. 类 Tmodem 中几个操作 Modem 的成员函数

(1) procedure TModem.InitModem

对 Modem 进行初始化。首先初始化端口，复位 Modem，然后初始化 Modem。

(2) function TModem.InitCommunication:Boolean

初始化通信。如果初始化失败超过指定的次数（初始化 Modem 的最多的次数），则终止本次初始化。

结果：初始化成功，返回 True，否则返回 False。

(3) procedure TModem.DialAModem(const strPhone:string)

用 Modem 拨号。

输入参数：strPhone 将要拨打的电话号码。

结果：拨打指定的电话号码。

(4) function TModem.AnalyzeModemResponseCode:integer

分析 Modem 的返回字符串，返回相应的常量以给出分析结果。

(5). procedure TModem.GetInfoFromModem(var strInput:string;const StrMode:string)

从 Moedm 中获得数据。

输入参数：

strInput 接收字符串的变量

StrMode 接收模式，0 - comInputModeText 文本模式

1 - comInputModeBinary 二进制模式

结果: 从 Modem 中获得字符串。

```
(6) function TModem.SendPackageIntoModem(
                                const aPackageSending:trPackageSending):Boolean
```

发送数据包。

输入参数: aPackageSending 将要发送的信息包。

返回值: True 表示发送成功;

False 表示发送不成功。

```
(7) procedure TModem.HangUpModem
```

将 Modem 挂机。首先将 Modem 切换到命令状态, 然后发送挂机命令。

结果: Modem 挂机。

2. 处理数据封装和解包的函数库 Packages

其中, Packages 为一个处理数据封装和解包的函数库, 本单元用到了 Packages 中的部分函数和说明。在第 9 章中的实例需要用到库单元 Packages, 特别说明如下:

```
const cnPackageLen=64;          //信息包长度
type
  tByteArray=array [1..cnPackageLen] of byte;
  trPackageSending =record
    //将要发送的信息包。所有生成的信息包, 先拷贝到此信息包中, 然后由程序
    //发送该信息包
    PackageGeneral:tByteArray;
    iLen:integer;
  end;
  trPackageReceived =trPackageSending; //接收到的信息包
  trPackageUnitsStatus=record      //列架工作状态信息包, 来自于监测单元
    iPhoneLen:integer;             //电话号码的长度
    strPhone:string[cnPhoneLen];  //监测单元的电话号码
    iUnitsNum:integer;             //监控的列架数目
    PackageUnitsStatus:tByteArray; //列架的工作状态
    strStatus:string[cnMaxNumStatusBytes]; //内容与 PackageUnitsStatus 相同, 只是
                                         //数据类型不同
    vStatus:Variant;              //内容与 PackageUnitsStatus 相同, 只是数据类型不同
    iHasBadUnit:integer;           //1 表示无坏单元
    iBadUnitsNum:integer;          //坏单元的数目
    iLen:integer;                  //信息包的长度
  end;
```

另外, 单元 cModem 中用到一个名为 MemoModemLog 的控件, 即一个 TMemo 控件。
bTerminateAllThreads 变量, 用于终止所有的线程, 在该单元中 bTerminateAllThreads 用在

InitCommunication 函数中，用来停止初始化 Modem。

3. 单元 cModem 详细说明

unit cModem;

//控制 Modem 的基本单元，提供对 Modem 进行操作的几乎所有功能

//本类使用了 Umanager 中的 MemoModemLog 控件和 bTerminateAllThreads 变量

interface

uses OleCtrls, MSCommLib_TLB, ComCtrls, Packages, Windows, Classes, SysUtils;

//常数声明

const cnModemLogFilePath='ModemLog\ModemLog'//日志文件所在路径及文件名

const cnModemLogFileMaxSize=8388608/2; //日志文件的最大大小

const cnReturn=chr(13)+chr(10); //回车换行键

const cnModemLogLen=4096; //日志的长度

const cnPackageStarter='START'; //数据包的开头修饰符

const cnModemResponseTimeOut=3000; //Modem 最大的响应时间，

//如果 Modem 响应时间超过指定的标准，则认为 Modem 出错

const cnWaitBeforeHangUp=1500; //在挂机之前的等待时间

const cnReceiveTimeOut=4000; //接收超时。如果在指定的时间内没能接收到
//足够的数据，则超时

const cnSendTimeOut=2400; //发送超时。如果在指定的时间内

//未能发送出足够的数据，则超时。初始化 Modem 的字符串常量

const

cnInitModemString='ATE1M0Q0S0=3V1X4&C1&D3&K0&S0'+cnReturn;//+Chr(13);

const cnInitModemMaxAmount=18;//初始化 Modem 的最多的次数，

//如果超过指定的次数，则终止对 Modem 的初始化

//Modem 响应常量。欲知 Modem 响应常量详细说明，请参见前面章节

const MODEM_CORRECT_EXECUTE=0; //AT 命令正确执行

const MODEM_INVALID_ATCOM =1; //无效的 AT 命令

const MODEM_CONNECTED =2; //已建立连接

const MODEM_CARRIER =3; //已检测到载波

const MODEM_NOCARRIER =4; //无载波

const MODEM_BUSY =5; //忙音

const MODEM_RING =6; //振铃

const MODEM_NODIALTONE =7; //无拨号音

const MODEM_UNKNOW_RETURN =10; //不能分辨的结果码

const MODEM_NO_ANSWER =11;//当拨一个不提供拨号音的系统时，

//未检测到无声信号。由拨号修饰符@开启)

const MODEM_NO_VALUE =12;//没有值

type

//声明一个类

TModem=class(TObject)

Private//私有属性

fbReceiveCompleted:Boolean; //True 表示接收到完整的信息包，完整的信息包
//不表示接收的信息包是正确的信息包，仅表示
//该信息包的长度是对的

fbConnected:Boolean; //True 表示与远端 Modem 连接成功，False 表示与
//远端 Modem 连接或连接失败

fstrRecBuffer:string; //在拨号时，初始化该变量
//在连接或拨号时出错，清空该变量。
//在完成数据发送或接收后，清空该变量。
//在接收数据包阶段，数据包先存入该变量，然后
//拷贝到 aPackageReceived

fbHasCarrier:Boolean; //True 表示载波存在，False 表示载波不存在。
//拨号后载波存在，如通信中断，可能造成载波丢失，
//挂机后载波不存在

fbCanNotCommunicate:Boolean; //True 表示不能通信，否则能够通信

fMSComm:TMSComm;

fslModemLog:TStringList;

fModemLogFilePath:string; //Modem 日志文件的路径

fbInitSuccessful:Boolean; //True 表示初始化 Modem 成功

filnitModemAmount:integer; //初始化 Modem 的次数

fstrPrefixion:string; //加在 Modem 日志上的前缀

protected

destructor Destroy;override;

public

aPackageReceived:trPackageReceived; //接收的信息包

strModemLog:string; //用于记录 Modem 的通信过程

constructor Create(aMSComm:TMSComm);

procedure InitModem;

procedure ShowModemInfo(const strMsg:string);

procedure ShowModemPackage(const aPackageReceived:trPackageReceived);

procedure ShowInfoIntoMainThread(const strMsg:string);

procedure ShowStatusPackageInfo;

procedure HangUpModem;

procedure GetInfoFromModem(var strInput:string;const StrMode:string);

procedure DialAModem(const strPhone:string);

procedure StoreIntoReceivePackage(const aReceiveByte:Byte);

procedure StoreIntostrRecBuffer(const strReceive:string);

```

function SendPackageIntoModem(
    const aPackageSending:trPackageSending):Boolean;
procedure SaveModemLog;
function HasResponseCode(const astrReceived:string;
    const aStr:string):Boolean;
function AnalyzeModemResponseCode:integer;
procedure ClearRecBuffer;
function InitCommunication:Boolean;
end;

implementation

uses uManager,Dialogs,forms;//包括 uManager 是为了使用其中的 MemoModemLog 控件,
    //即一个 TMemo 控件和 bTerminateAllThreads 变量
//清空接收缓冲区 fstrRecBuffer
procedure TModem.ClearRecBuffer;
begin
    fstrRecBuffer:="";
end;

//判断 astrReceived 是否含有指定的字符串 aStr, 如有, 返回 True, 否则返回 False
//输入参数: astrReceived, 表示可能含有指定字符串的数组
//          aStr: 指定的字符串
function TModem.HasResponseCode(const astrReceived:string;
    const aStr:string):Boolean;
begin
    //调用 function AnsiPos(const Substr, S: string): Integer
    if AnsiPos(aStr,astrReceived)>0 then
        Result:=True
    else Result:=False;
end;

//分析 Modem 的返回字符串, 返回相应的常量以给出分析结果
//输入参数: aLocalPackageReceived, 表示欲分析的数组, 其中含有 Modem 返回字符串
function TModem.AnalyzeModemResponseCode:integer;
begin
    { const MODEM_CORRECT_EXECUTE=0;//AT 命令正确执行
      const MODEM_INVALID_ATCOM   =1;//无效的 AT 命令
      const MODEM_CONNECTED       =2;//已建立连接

```

```

const MODEM_CARRIER      =3;//已检测到载波
const MODEM_NOCARRIER    =4;//无载波
const MODEM_BUSY          =5;//忙音
const MODEM_RING          =6;//振铃
const MODEM_NODIALTONE    =7;//无拨号音
const    =10;//无知的结果码}
GetInfoFromModem(fstrRecBuffer,'Text');
Result:=MODEM_UNKNOWN_RETURN;
if HasResponseCode(fstrRecBuffer,'OK') then
    Result:=MODEM_CORRECT_EXECUTE;
if HasResponseCode(fstrRecBuffer,'ERROR') then
    Result:=MODEM_INVALID_ATCOM;
if HasResponseCode(fstrRecBuffer,'CONNECT') then
    Result:=MODEM_CONNECTED;
if HasResponseCode(fstrRecBuffer,'NO CARRIER') then
    Result:=MODEM_NOCARRIER;
if HasResponseCode(fstrRecBuffer,'BUSY') then
    Result:=MODEM_BUSY;
if HasResponseCode(fstrRecBuffer,'RING') then
    Result:=MODEM_RING;
if HasResponseCode(fstrRecBuffer,'NO DIALTONE') then
    Result:=MODEM_NODIALTONE;
if HasResponseCode(fstrRecBuffer,'NO ANSWER') then
    Result:=MODEM_NO_ANSWER;
ClearRecBuffer;
end;
//释放对象申请的空间,同时将 Modem 的通信日志存盘
destructor TModem.Destroy;
begin
    //释放申请的内存
    SaveModemLog;
    fslModemLog.Free;
    inherited Destroy;
end;

//创建一个 Modem 对象,初始化相关的变量
constructor TModem.Create(aMSComm:TMSComm);
begin
    //设置对应的 MSComm 控件

```

```

fMSComm:=aMSComm;
fstrPrefixion:='串口'+IntToStr(fMSComm._CommPort)+'';
fbCanNotCommunicate:=False;
fbReceiveCompleted:=False;
fbHasCarrier:=False;
//清空接收缓冲区
fstrRecBuffer:='';
fbConnected:=False;
strModemLog:='';
fslModemLog:=TStringList.Create;
//vTmp:=VarArrayCreate([0,127],varByte);
//设置 Modem 通信日志文件的路径及文件名
fModemLogFilePath:=ExtractFilePath(ParamStr(0));
fModemLogFilePath:=fModemLogFilePath+
    cnModemLogFilePath+'COM'+
    IntToStr(fMSComm._CommPort)+'.txt';
inherited Create;
end;

```

```

//对 Modem 初始化
//结果:对 Modem 进行初始化。首先初始化端口,
//复位 Modem;然后初始化 Modem
procedure TModem.InitModem;
var dwModemResStart,dwModemResEnd:DWord;
//Modem 响应的开始时间, Modem 响应的结束时间
begin
    ShowModemInfo(fstrPrefixion+'初始化 Modem!');
    fbCanNotCommunicate:=False;
    try
        try
            if fMSComm.PortOpen then
                begin
                    fMSComm.PortOpen:=False;
                    Sleep(1000);
                end;
            if not fMSComm.PortOpen then
                fMSComm.PortOpen := True;
        except
            //处理端口初始化的错误信息

```

```

on E: Exception do
begin
    ShowModemInfo(fstrPrefixion+E.Message); // 查 E.HelpContext
    ShowModemInfo(fstrPrefixion+'初始化 Modem 失败! ');
    fbCanNotCommunicate:=True;
    exit;
end;
end;
//等待 Modem 的正确响应
dwModemResStart:=GetTickCount;
while fMSComm.DSRHolding<>True do
begin
    dwModemResEnd:=GetTickCount;
    //Modem 响应超时, Modem 可能有问题
    if ((dwModemResEnd-dwModemResStart)>cnModemResponseTimeOut) then
    begin
        //处理来自于 Modem 的错误
        ShowModemInfo(fstrPrefixion+'Modem 响应失败, Modem 初始化失败! ');
        fbCanNotCommunicate:=True;
        Exit;
    end;
end;
//现在, DSRHolding=TRUE, 即可以向 Modem 发命令
fMSComm.InBufferCount:=0; //清空接收缓冲区
fMSComm.InputLen:=0; //Input 读取整个缓冲区的内容
fMSComm.RThreshold:=1; //每次接收到字符即产生 OnComm 事件
fMSComm.DTREnable:=True; //数据终端准备好
fMSComm.RTSEnable:=True; //请求发送

Sleep(1100);
fMSComm.Output:='+++'+cnReturn;
Sleep(1100);
fMSComm.Output := cnInitModemString;
//ATE1M0Q0S0=3V1X4&C1&D3&K0&S0'+cnReturn;//+Chr(13)
//Modem AT 命令说明
//E1 在命令状态打开字符响应
//M1 扬声器打开, 直到检测到载波才关闭
//Q0 Modem 返回结果码
//s0=2 开始应答前振铃 2 次

```

```

//S9=10 选择载波检测响应时间
//V1 以字符形式显示结果码
//X4 提供基本呼叫进程结果码、连接速率、忙音信号检测和拨号音检测
//&A0 当设置为自动应答方式时，以应答方式建立连接
//&C1 跟踪数据载波 (Trace Data Carrier)
//&D3 当 DTR 发生从开到关的转换时，复位
//&K0 关闭本地流量控制
//&S0 长开启 DSR 信号
//确保 Modem 给出响应"OK"
sleep(1200);
except
  on E: Exception do
  begin
    ShowModemInfo(fstrPrefixion+E.Message);
    ShowModemInfo(fstrPrefixion+'初始化 Modem 失败! ');
    fbCanNotCommunicate:=True;
    Exit;
  end;
end;
//初始化变量
strModemLog:="";
fbHasCarrier:=False;
fstrRecBuffer:="";
fbConnected:=False;
fbReceiveCompleted:=False;
aPackageReceived.iLen:=0;
fbCanNotCommunicate:=False;
end;

//处理 Modem 的日志信息
//输入参数: strMsg 将存入 Modem 日志的信息
//结果: 如果 Modem 的日志过长, 将 Modem 日志清空, 存入新的信息
//否则, 将信息追加到日志
procedure TModem.ShowModemInfo(const strMsg:string);
var strTmp:string;
begin
  try
    strTmp:=strMsg;
    strTmp:=DateTimeToStr(now)+'-'+strTmp;

```

```

ShowInfoIntoMainThread(strTmp);
strTmp:=strTmp+cnReturn;
if length(strModemLog+strTmp)<cnModemLogLen then
    strModemLog:=strModemLog+strTmp
else
begin
    //在清空日志信息之前，先存盘
    SaveModemLog;
    strModemLog:="";
    strModemLog:=strTmp+cnReturn;
end;
except
end;
end;

//将信息显示到主线程中
//输入参数: strMsg, 待显示的信息
procedure TModem.ShowInfoIntoMainThread(const strMsg:string);
begin
    frmManager.MemoModemLog.Lines.Add(strMsg);
    //将信息显示到 frmManager 中含有叫 MemoModemLog 的 TMemo 控件,
    //读者可以修改为自己的显示方法
end;

//显示数据包的内容, 以字符串形式
//输入参数: aPackageReceived, 待显示的数据包
procedure TModem.ShowModemPackage(const aPackageReceived:trPackageReceived);
var strTmp:string;
    i:integer;
    iTmp:integer;
begin
    strTmp:="";
    for i:=1 to aPackageReceived.iLen do
    begin
        iTmp:= aPackageReceived.PackageGeneral[i];
        strTmp:=strTmp+" "+format('%d',[iTmp]);
        //IntToStr(aPackageReceived.PackageGeneral[i]);
    end;
    ShowModemInfo(fstrPrefixion+'10 进制: '+strTmp);

```


end;

//将 Modem 挂机。首先将 Modem 切换到命令状态，然后发送挂机命令

//结果：Modem 挂机

procedure TModem.HangUpModem;

begin

 ShowModemInfo(fstrPrefixion+'断开与监测单元的连接并挂机!');

 try

 if (fMComm.PortOpen) then

 begin

 fMComm.Output:="";

 fMComm.OutBufferCount:=0; //清空输出缓冲区

 fMComm.InBufferCount:=0; //清空输入缓冲区

 fMComm.SThreshold:=1;

 fMComm.RThreshold:=1;

 Sleep(1100);

 fMComm.Output:='+++ '//+cnReturn;

 Sleep(1100);

 fMComm.Output:='ATH'+cnReturn; //发送关机命令

 fMComm.PortOpen:=False; //关闭串口

 end;

 fMComm.DTREnable:=False; //数据终端未准备好

 fMComm.RTSEnable:=False; //关闭请求发送

 except

 on E: Exception do ShowModemInfo(fstrPrefixion+E.Message);

 end;

 try

 SaveModemLog;

 finally

 end;

 strModemLog:="";

end;

//从 Modem 中获得数据

//输入参数

//strInput 接收字符串的变量，StrMode 接收模式，0 - comInputModeText 文本模式

//1 - comInputModeBinary 二进制模式。结果：从 Modem 中获得字符串

procedure TModem.GetInfoFromModem(var strInput:string;const StrMode:string);

var vTmp:variant;

```

    ovTmp:oleVariant;
    i:integer;
    iReceived:integer;
    bTmp:byte;
begin
    try
        if (fMSComm.PortOpen) then
            begin
                //0 - comInputModeText
                //1 - comInputModeBinary
                if strMode='Text' then
                    begin
                        fMSComm.InputMode:=comInputModeText;
                        strInput:=fMSComm.input;
                    end
                else
                    begin
                        //设置输入模式
                        fMSComm.InputMode:=comInputModeBinary;
                        //等待 Modem 的响应
                        Sleep(20);
                        iReceived:=fMSComm.InBufferCount;
                        ovTmp:=fMSComm.Input;
                        vTmp:=VarArrayCreate([0,127],varByte);
                        vTmp:=ovTmp;
                        strInput:="";
                        //将接收到的数据转换为字符串形式
                        for i:=0 to iReceived-1 do
                            begin
                                bTmp:=vTmp[i];
                                strInput:=strInput+Chr(bTmp);
                            end;
                        //设置输入模式
                        fMSComm.InputMode:=comInputModeText;
                    end;
                end;
            end;
        //if (fMSComm.PortOpen) then
    except
        on E: Exception do ShowModemInfo(fstrPrefixion+E.Message);
    end;
end;

```

```

//设置输入模式
fMComm.InputMode:=comInputModeText;
end;

//用 Modem 拨号。输入参数: strPhone 将要拨打的电话号码
//结果: 拨打指定的电话号码
procedure TModem.DialAModem(const strPhone:string);
begin
  try
    if fMComm.PortOpen then
      fMComm.Output:='ATDT'+strPhone+cnReturn
    else
      Raise Exception.Create('端口未打开! ');
  except
    on E: Exception do ShowModemInfo(fstrPrefixion+E.Message);
  end;
  aPackageReceived.iLen:=0;//初始化接收缓冲区
end;

//将 strReceive 存入到接收缓冲区
//输入参数: strReceive。结果: strReceive 存入到接收缓冲区 strRecBuffer
procedure TModem.StoreIntostrRecBuffer(const strReceive:string);
var iPos:integer;
    i,iStart:integer;
begin
  fstrRecBuffer:=fstrRecBuffer+strReceive;
  iPos:=pos(cnPackageStarter,fstrRecBuffer);
  if iPos>0 then
    begin
      //如果已经接收到指定长度的数据包
      if (Length(fstrRecBuffer)-iPos>=(cnPackageLen+Length(cnPackageStarter)-1)) then
        begin
          //拷贝数据包到接收缓冲区
          iStart:=iPos+Length(cnPackageStarter);
          for i:=iStart to iStart+cnPackageLen-1 do
            begin
              StoreIntoReceivePackage(Ord(fstrRecBuffer[i]));
            end;
          //将数据拷贝走后, 清空接收缓冲区
        end;
      end;
    end;
end;

```

```

        fstrRecBuffer:="";
    end;
end;
end;

//将数据存入接收数据包
//输入参数: aReceiveByte 将要存入的数据
//结果: 数据存入指定的数据包
procedure TModem.StoreIntoReceivePackage(const aReceiveByte:Byte);
begin
    //aPackageReceived should be initilized when receiving a byte firstly;
    aPackageReceived.PackageGeneral[aPackageReceived.iLen+1]:=aReceiveByte;
    aPackageReceived.iLen:=aPackageReceived.iLen+1;
end;

//发送数据包。Parameter:aPackageRSending 将要发送的信息包
//返回值: True 表示发送成功, False 表示发送不成功
function TModem.SendPackageIntoModem(
                                const aPackageSending:trPackageSending):Boolean;
var dwSendStart,dwSendEnd:DWord;
    //发送开始时间, 发送结束时间
    strTmp:string;
    ovTmp:Olevariant;
    i:integer;
    vTmp:Variant;
begin
    Result:=True;
    //获取发送开始时间
    dwSendStart:=GetTickCount;
    //处理 Modem 响应问题: 端口打开问题、载波丢失问题、Modem 无法响应问题
    while Not (fMSComm.PortOpen and fMSComm.CDHolding and fMSComm.CTSHolding) do
    begin
        dwSendEnd:=GetTickCount;
        if (dwSendEnd-dwSendStart>cnSendTimeOut) then
        begin
            if not fMSComm.PortOpen then ShowModemInfo(fstrPrefixion+'Modem 端口未打
                                                    开! ');
            if not fMSComm.CDHolding then ShowModemInfo(fstrPrefixion+'载波丢失! ');
            if not fMSComm.CTSHolding then ShowModemInfo(fstrPrefixion+'Modem 无法响

```

应! ');

```

    ShowModemInfo(fstrPrefixion+'Modem 不能发送数据! ');
    Result:=False;
    Exit;
end;
end;
try
    if (fMSComm.PortOpen and fMSComm.CDHolding and fMSComm.CTSHolding) then
    begin
        //在发送数据之前, 对数据进行转换
        //先从 string 转换为 Variant, 再从 Variant 转换为 oleVariant
        strTmp:=cnPackageStarter;
        //发送数据包的修饰符, 将发送的数据包进行转换, 先从 string 转换为 Variant
        vTmp:=strTmp;
        //从 Variant 转换为 oleVariant
        ovTmp:=vTmp;
        //发送数据包的修饰符
        fMSComm.output:=ovTmp;
        vTmp:=VarArrayCreate([1,cnPackageLen],VarByte);
        for i:=1 to aPackageSending.iLen do
        begin
            vTmp[i]:=aPackageSending.PackageGeneral[i];
        end;
        ovTmp:=vTmp;
        //发送数据包
        fMSComm.output:=ovTmp;
        //处理发送超时
        dwSendEnd:=GetTickCount;
        while true do
        begin
            if fMSComm.OutBufferCount<=0 then
            begin
                Result:=True;
                break;
            end
            else
            begin
                //条件为 fMSComm.OutBufferCount>0
                if not fMSComm.CDHolding then
                begin

```

```

        ShowModemInfo(fstrPrefixion+'数据发送过程中, 载波丢失! ');
        Result:=False;
        Exit;    //从当前的 procedure 退出
    end;
    if (dwSendEnd-dwSendStart>cnSendTimeOut) then
    begin        //处理发送超时
        ShowModemInfo(fstrPrefixion+'Modem 发送数据超时! ');
        Result:=False;
        Exit;    //从当前的 procedure 退出
    end;
end;
end;    // while true do 语句结束
end; //if (fMSComm.PortOpen and fMSComm.CDHolding and fMSComm.CTSHolding)
except
    on E: Exception do
    begin
        ShowModemInfo(fstrPrefixion+E.Message);
        Result:=False;
    end;
end;
end;

//存储通信日志。如果文件大小超过 cnModemLogFileMaxSize, 则将原来的
//文件改名为 fModemLogFilePath+当前日期.txt
procedure TModem.SaveModemLog;
var hTextFile:TextFile;
    sr:TSearchRec;
    strNewFileName:string;
    //hTextFile:Tfilestream;
begin
    try
        //如果文件大小超过 cnModemLogFileMaxSize, 则将原来的
        //文件改名为 fModemLogFilePath+当前日期.txt
        if FileExists(fModemLogFilePath) then
        begin
            try
                if FindFirst(fModemLogFilePath,faAnyFile,sr)=0 then
                begin
                    if sr.Size>cnModemLogFileMaxSize then

```

```

begin
    strNewFileName:=FormatDateTime("截止到"yyyy"年"mm"月"dd"日"hh"时
                                   "mm"分"ss"秒",now);
    strNewFileName:=strNewFileName + '的日志文件.txt';
    strNewFileName:=
        ExtractFilePath(ParamStr(0))+ 'ModemLog\' + strNewFileName;
    {调用 function RenameFile(const OldName, NewName: string): Boolean;}
    RenameFile(fModemLogFilePath, strNewFileName);
end;
end;
finally
    FindClose(sr); //释放资源
end;
end;

AssignFile(hTextFile, fModemLogFilePath);
//如果通信日志文件不在, 则生成一个新的文件
if not FileExists(fModemLogFilePath) then
    ReWrite(hTextFile);
Append(hTextFile);
WriteLn(hTextFile, strModemLog); //Length(strModemLog+cnReturn))
except
begin
    ShowInfoIntoMainThread('写日志文件出错!, 请清除一部分通信日志!! ');
    try
        //存盘, 并关闭文件
        Flush(hTextFile);
        CloseFile(hTextFile);
    except
        exit;
    end;
    exit;
end;
end;
Flush(hTextFile); //存盘, 并关闭文件
CloseFile(hTextFile);
end;

//TModem.InitCommunication 函数作用为初始化通信。如果初始化失败超过指定的
//次数(初始化 Modem 的最多的次数), 则中止本次初始化

```

//结果: 初始化成功, 返回 True, 否则返回 False

本函数的流程如图 4.5 所示。

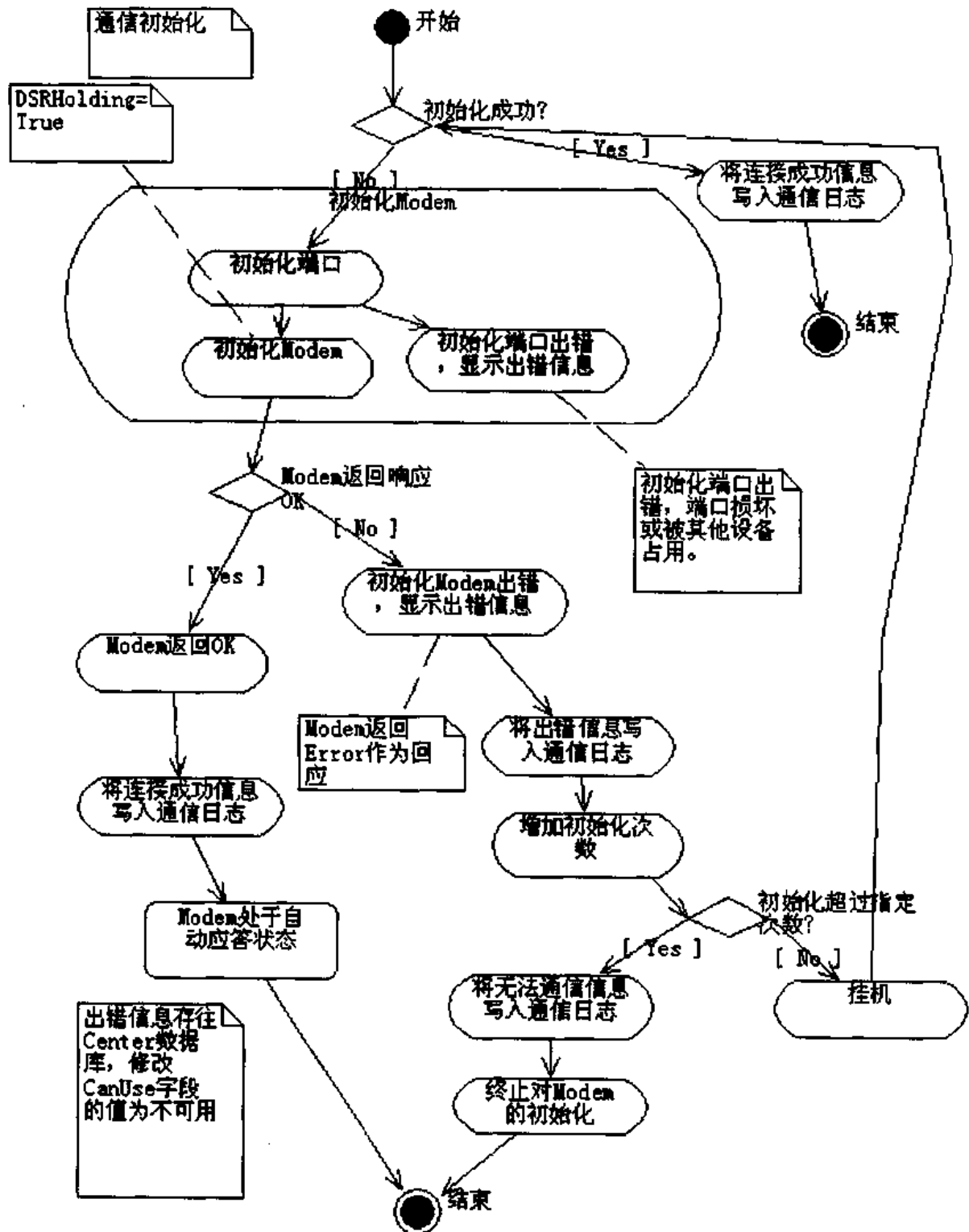


图 4.5 本函数的流程


```

function TModem.InitCommunication:Boolean;
begin
    Result:=False;
    fbInitSuccessful:=False;
    while (not fbInitSuccessful) do
    begin
        try
            InitModem;    //初始化后, Modem 处于自动应答状态
        { const MODEM_CORRECT_EXECUTE=0; //AT 命令正确执行
          const MODEM_INVALID_ATCOM  =1;  //无效的 AT 命令
          const MODEM_CONNECTED      =2;   //已建立连接
          const MODEM_CARRIER       =3;   //已检测到载波
          const MODEM_NOCARRIER     =4;   //无载波
          const MODEM_BUSY           =5;   //忙音
          const MODEM_RING           =6;   //振铃
          const MODEM_NODIALTONE     =7;   //无拨号音
          const MODEM_UNKNOW_RETURN  =10;  //无知的结果码
          const MODEM_NO_ANSWER      =11;  //当拨一个不提供拨号音的系统时,
              //未检测到无声信号(由拨号修饰符@开启) }
            //判断 Modem 的响应, 如果返回的字符串非 OK, 则错误
            if (AnalyzeModemResponseCode<>MODEM_CORRECT_EXECUTE) then
            begin
                fbInitSuccessful:=False;
                Inc(fiInitModemAmount);
                ShowModemInfo(fstrPrefixion+'Modem 初始化失败! ');
                //如果超过指定的次数(初始化 Modem 的最多的次数), 则终止本次初始化
                if fiInitModemAmount>cnInitModemMaxAmount then
                begin
                    fbInitSuccessful:=False;
                    ShowModemInfo(fstrPrefixion+'Modem 初始化失败!!!, 无法与 Modem
                        通信!!! ');
                    break;
                end
            else
            begin
                HangUpModem;
                if bTerminateAllThreads then break;
                //终止所有的活动线程, 停止初始化 Modem
                ShowModemInfo(fstrPrefixion+'开始第'+IntToStr(fiInitModemAmount)

```

```

        + '次重新初始化 Modem');

        Application.ProcessMessages;
        //中断 Application 的执行使 Windows, 处理消息队列
        Sleep(500);
    end;
end
else
begin
    //Modem 正确执行了命令
    fbInitSuccessful:=True;
    fiInitModemAmount:=0;
    Result:=True;
    ShowModemInfo(fstrPrefixion+'Modem 初始化成功! ');
end;
except
try
    ShowModemInfo(fstrPrefixion+'???? Modem 初始化过程中出现异常!, 重新初始化! ');
except
end;
end;
end;    // while (not terminated) and (not fbInitSuccessful) do
end;

```

//笔者参与的项目中要求实现的一部分功能, 非对 Modem 操作的函数

//如果读者不需要, 此函数可以去掉。注意, 第9章中的实例需要该功能

//显示状态信息包中的内容。状态信息包中的内容存于 aPackageUnitsStatus 中,

//因此本过程显示 aPackageUnitsStatus 中的内容

```

(*  trPackageUnitsStatus=record                //列架工作状态信息包, 来自于监测单元
    iPhoneLen:integer;                        //电话号码的长度
    strPhone:string[cnPhoneLen];             //监测单元的电话号码
    iUnitsNum:integer;                        //监控的列架数目
    PackageUnitsStatus:tByteArray;           //列架的工作状态
    strStatus:string[cnMaxNumStatusBytes];   //内容与 PackageUnitsStatus 相同, 只是
                                                //数据类型不同
    iHasBadUnit:integer;                      //1 表示无坏单元
    iBadUnitsNum:integer;                     //坏单元的数目
    iLen:integer;                             //信息包的长度
end;
aPackageUnitsStatus *)

```

```

procedure TModem.ShowStatusPackageInfo;
const cnSpace='
var strTmp:string;
begin
    strTmp:='信息包来自于串口'+IntToStr(fMComm.CommPort)+cnReturn;
    strTmp:=strTmp+'信息包内容为'+cnReturn;
    strTmp:=strTmp+cnSpace+'监测单元的电话号码为: '
                +aPackageUnitsStatus.strPhone+cnReturn;
    strTmp:=strTmp+cnSpace+'监测单元的列架数为: '
                +IntToStr(aPackageUnitsStatus.iUnitsNum)+cnReturn;
    strTmp:=strTmp+cnSpace+'监测单元的坏单元数目为: '
                +IntToStr(aPackageUnitsStatus.iBadUnitsNum);
    ShowModemInfo(strTmp);
end;

end.

```

4.4 使用技巧

1. 在拨通对方主机电话后，再拨分机号

问题：如何利用已经安装的 Modem，在拨通对方主机电话后，再拨分机号？（例如：对方的电话号码是 7931666 转 1234）

回答：如果对方的分机是自动切换，比如在拨通后继续按分机号，可以使用 ATDT7931666,1234 来拨号，其中的逗号表示拨号间稍微等待一下，读者可以根据实际情况多加几个逗号。如果对方是人工切换，那就别费事了（如果读者的 Modem 是语音 Modem，尽管理论上也有办法，但过于麻烦，而且不易成功）。

2. 如何编写挂断 Modem 的程序

问题：如何编写挂断 Modem 的程序？

回答：如果读者使用的是 MComm 控件，可以设置 PortOpen:=False。或者，当 Modem 处于接收命令的状态时，向 MComm 的 Output 发出 ATH 命令。否则，先向 MComm 的 Output 发出+++命令，等待大概 1 秒钟后，再发 ATH 命令。

3. 如何通过 Modem 读取

问题：如何通过 Modem 读取电话线上的电话机数字键发出的 DTMF 信号，以实现 BITWARE 中用电话机对语音信箱的操作？

回答：标准 Modem 中大概有 30 个接口寄存器，读者对 Modem 是无法直接访问这些寄

存器的，只有通过 AT 命令，由 Modem 来完成一些特定的功能。关于 DTMF 检测（也就是对方电话号码的检测），要求该 Modem 芯片必须具有 CALLER ID 的功能，至于 AT 命令中是否有检测 CALLER ID 功能的命令，需要读者去查阅相关的最新资料，因为以前的 AT 命令不包括这个功能，计算机对 Modem 的唯一访问方式就是 AT 命令。（建议：一般在 Modem 的手册上会有 AT 指令集）。

目前的 Modem 一般都不具备检测 DTMF 功能，所以一般需要另外购买硬件（如电话语音卡）。

4. 同时从微机的两个串口接收信息

问题：一个告警信息监控程序，该程序需要同时从微机的两个串口接收信息，并进行过滤翻译处理（需要综合多行信息），并将结果显示在屏幕上，想用 MSComm 控件编写，请问这是否用到多任务？该怎样处理这个问题？是用 MSComm 的“事件驱动”方式还是“查询”方式？问题的关键在于要同时处理两个串口的数据接收，两个串口收到数据的处理方式是不同的。

回答：当然用事件驱动，如果读者的两个处理过程不是太复杂和太过于占用时间使用查询也行。但还是事件驱动快。不用多线程，只用事件驱动就足够了。

在界面上放两个串口控件，再使用 PORT=1 和 2（PORT 应根据读者的具体情况设定），即可对两个串口进行控制，接收和发送均可。

5. 通过 Modem 传递语音

问题：用 MSComm 控件通过 Modem 拨到对方的电话上，并放一个语音到对方的电话上。

回答：首先读者的 Modem 必须有语音功能。然后，将欲传送的语音以 WAV 方式录制好。语音 Modem 应该支持 AT+V 指令（个别 Modem 使用 AT#V 指令），利用 Windows API 中 waveOutOpen、waveOutWrite 等函数向 Modem 设备输出 WAV 声音。在调用 API 时要使用 Modem 的设备 ID，读者可以从 MSComm 控件的 CommID 属性获得。

6. 如何在拨号时等待 2~3 秒后再拨出号码

问题：如何能够实现先拨一个号码，在不断线的情况下，等待 2~3 秒，再拨出号码？

回答：在两个号码中间加上“，”，如果等待时间不够长，可以再加几个。如：ATDT12345678,90。

7. 如何通过串口接收数值 0

问题：先把串口通信控件的“InputMode”设置为“1-comInputModebinary”此参数为以二进制方式接收。

回答：“NullDiscard”设置为 false，此参数允许 0H 传输到缓冲区。“Rthreshold”设置为 1，此参数为当收到任一字符时均产生“OnComm”事件。单元 cModem 中 GetInfoFromModem 函数有详细的代码。

8. 如何检测对方是否已经接听或对方是否挂断

问题：在设计一个用 Modem 与电话通信的软件时，用的是 MSComm 控件，但现在无法

检测对方是否已经接听（即提起电话）或对方是否挂断（放下电话）？

回答：检查 MSComm 的属性 CDHolding，如果为 True 表示在线，否则表示线路已断开。

9. 如何获得系统已安装的 Modem 的端口号

问题：如何获得系统已安装的 Modem 的端口号？

回答：一个办法是用 MSComm 控件打开端口，如果打开 Com1 错误，Modem 就在 Com2 上。

另一个办法是读取注册表的 HKEY_LOCAL_MACHINE\ SYSTEM\ CurrentControlSet\ Service\ Class\ Modem，这下面的子键列举了系统的 Modem 情况，其中的 AttachedTo 表明读者的 Modem 到底在哪个端口。

10. 检测 Modem 是否正在使用

问题：检测 Modem 是否正在使用？

回答：简单的办法是使用 Visual Basic 的 MSComm 控件。可以使用 PortOpen 来打开 Modem。如果 Modem 被其他软件占用，可能产生错误，错误号是 8005（Port already open）或 8010（The hardware is not available （locked by another device））。

11. 调用 MSComm.PortOpen:=True 也出错，显示“串口已打开”

问题：在 Delphi 里安装了 MSComm 控件，在 Form1 里用 MSComm.PortOpen:= TRUE 打开串口 1 后，Form1 调用 Form2，在 Form2 里又放置了 MSComm 控件，但在 Form2 里提示“无法对未知的口操作”，如在 Form2 里再调用 MSComm.PortOpen:=TRUE 也出错，显示“串口已打开”。

回答：检测 Modem 是否正在使用同一个串口同时只能被打开一次，读者已经在 Form1 中打开串口了，所以在 Form2 中无法再打开了。可用下面两个办法：

（1）在调用 Form2 前，关闭串口。

（2）Form1 的控件不关闭串口，在 Form2 中不使用 MSComm 控件，而是调用 Form1 的控件进行输入输出。

12. 从 MSComm 的缓冲区取出一部分数据，数据在缓冲区是否还存在

问题：MSComm 控件的接收缓冲区是如何管理的？如果有一个外设不断随机往接收缓冲区发数据，取出一部分数据，这部分数据在缓冲区中还存在吗？

回答：当然不在了。另外按上面说的情况，要用一个字节一个字节由程序读入的方式工作，即 InputLength=1 这种方式。

13. MSComm 无法稳定完整接收数据

问题：DOS 操作系统通过串口发送字符数据（连续不断），Windows 98 下 MSComm 无法稳定完整接收数据（特别是在 Windows 98 下同时有其他任务在执行时，系统处理变慢时）？

回答：Windows 98 是一个多任务系统，系统调度采用优先级竞争的策略。串口不能完整接受数据是因为 CPU 时间片被其他线程抢走。基于这一点，可采用 3 种解决方法：

（1）采用硬件缓冲机制。

(2) 改善通信协议, 确保数据传输的完整性与正确性。

(3) 提高通信程序的运行优先级, 最好为实时优先级。可通过调用 API 函数实现。
建议采用方法 (1)、(2)。

14. 接收的数据少于发送的数据

问题: 接收的数据少于发送的数据?

回答: 如果通过 MSComm 控件一次性传送较多的二进制数据, 那么, 很可能收到的数据不足。例如在设置为 2400bit/s 传输率的情况下, 一次性可以传输 2048 个字符数据, 那么在大多数情况下一次只能收到 1200 个字符左右, 这是因为新版的 MSComm32.OCX 中存在一个影响传输二进制数据的臭虫 (bug), 注意这不是特性。

32 位 Windows API 函数 (以下简称 API) 使用了几个用 COMMTIMEOUTS 结构表示的限时变量, WriteTotalTimeOutConstant 即是其中的一个, 它被 Windows 内部设定为 5000 (即 5 秒), 这个常量决定了在通信驱动程序停止传输之前, 花费在发送缓冲区中数据的时间的长短。5 秒钟意味着通信速度为 1200bit/s 情况下仅能发送 600 个字符, 2400bit/s 情况下仅能发送 1200 个左右的字符。事实上, 在一个缓冲区内一次性发送更多的数据是非常可能的。这个 bug 同样也能引发问题, 甚至在高速串口通信情况下, 即使系统在使用流控制, 无论是软件流 (Xon/Xoff) 还是硬件流 (CTS/RTS)。假如数据在发送缓冲区中时流控制停止了传输, 如果停止时间超过 5 秒钟, 则数据就会丢失。在某些环境下, 5 秒钟可能相当短, 不过也不必担心, Visual Basic 5.0/6.0 版本的 MSComm 控件有一个新增的重要的属性称为 CommID, CommID 指的是当串口被打开时, 被 API 所调用的串口句柄或称标志, 这也意味着能利用 API 接口函数去修改这个常量。每次串口关闭后, Windows 会自动将之恢复为 5000, 所以, 每次打开串口后需要重新设定 API 声明。

15. 如何发送大于 128 的字符数据

问题: 如何发送大于 128 的字符数据?

回答: 在通信程序中, 以单字符方式逐个发送数据时, 每一个数据范围为 0~255 (即十六进制的 00 到 FF)。在单字符版本的英文 Windows 95、DOS 版的 Basic 程序或 Delphi 中, 只需要将相应的数据转换成相应的字符发送到通信端口即可。但在中文 Windows 95/98 下却行不通, 假设在中文 Windows 95/98 下运行以下程序:

```
var I: integer;
for I:=0 to 255 do
begin
    MSComm1.Output:=Chr(I);
end;
```

希望在接收端得到预期的 0 到 255 之间的数据, 结果却是: 前 129 个数据接收正确, 为 0 到 128, 后面 127 个数据为 126 个 0 和一个 255。造成这种结果的原因在于中文 Windows 使用的是双字节字符集 (DBCS) 系统。DBCS 系统使用 0~128 之间的数字表示 ASCII 字符, 大于 128 的数字仅作为前导字符, 它只是显示是一个非拉丁语系的字符, 而并不代表实际意义。上述程序在调用 Chr 函数时用到了 DBCS 字符集, 因此产生了此类错误。那么, 如何发送大于 128 的数据呢? 请参考单元库 cModem 中函数 GetInfoFromModem, 其提供详细的代

码。

16. 如何发送 0 字符 (00H, NULL)

问题：如何发送 0 字符 (00H, NULL)？

回答：在 Visual C++ 中使用串口控件发送 0 字符有些麻烦，但在 Delphi 中只要注意以下两点即可：

(1) 设置 MSComm 控件的属性 NullDiscard 为 False。

(2) 使用二进制接收，即用 `MSComm1.InputMode:=comInputModeBinary` 便可以解决问题。

17. 如何发送中文字符串 (DBCS) 字符

问题：如何发送中文字符串 (DBCS) 字符？

回答：Visual Basic 5.0/6.0 的各种参考书上均指明 MSComm 通信控件不能发送或接收双字节字符集系统 (DBCS) 的二进制数据，这对于我国及亚洲一些使用 DBCS 字符集的国家是一大遗憾。但是在实践中发现，用 MSComm 控件也可以发送中文字符，具体方法有两种：

(1) 直接发送

直接发送即把中文字符等同于英文字符。如：`MSComm1.output='这是一行中文数据！'`，但这种方法发送的中文数据不能太长，发送缓冲区和接收缓冲区的大小需设定为中文字符的 2 倍以上，而且发送与接收系统所处的操作系统版本最好要一致，否则会出现接收或发送缓冲区溢出之类的错误。这种方法可用于要求不太高的场合。

(2) 间接发送

在发送端将汉字或字符转换为机器内码或区位码数据数组，然后将转换后的数据发送到串口，在接收端接收到数据后，按照相反的顺序将得到的数据转换为相应的汉字或字符。在转换过程中，要用到位运算，如取得汉字的内码后需要将高字节和低字节分开，而 Visual Basic 5.0/6.0 中并没有提供此类函数，Delphi 提供了求整数高、低字节的函数，函数分别为 `Hi(x)`、`Lo(x)`。

18. 如何用单机进行通信测试

问题：如何用单机进行通信测试？

回答：通常在写好了通信程序后需要两台 PC 或一台 PC、一台单片机，将通信口连接后进行测试，但很多时候因条件限制仅有单台 PC 机，测试项目很简单，那么能否测试呢？当然可以，而且方法也很简单。对于九针的串口，找一个废弃的串口鼠标，剥开鼠标线，将连接 2、3 针的线对接即可。对于 25 针的串口，可找一枚曲别针（最好有塑料外套的）将它扯直，剥去两头的塑料后在两头各弯一个圆圈，中间对折后直接套接在串口的 2、3 针上即可。如果担心不够安全，则可以将 5 针接地。

本章小结

本章详细介绍了 MSComm 通信控件的方法、属性及事件。MSComm 控件将通信的大部分操作都封装在控制内部，高层通信应用程序只要获取和设置相应的 MSComm 控件属性即可，大大地简化了编程。它有查询和中断两种编程方式，其中通过 OnComm 事件实现中断编程。

本章还详细介绍了一个控制 Modem 的基本库单元 cModem，以及一些使用技巧。

第 5 章 线程开发

本章主要内容：

- 进程和线程
- 线程的同步
- 线程的优先级
- TThread 对象
- 多线程实例

目前大多数计算机采用的是冯·诺依曼结构，它的特点为顺序处理，一个处理器同一时刻只能处理一件事情。Win32 系统采用新的任务调度策略，它将一个进程划分为多个线程，每个线程轮流占用 CPU 的运算时间，操作系统不断地将线程挂起、唤醒、再挂起、再唤醒，如此反复，由于现在 CPU 的速度很快，给人的感觉是多个线程在并行运行。

在 Windows NT 任务管理器的进程项中，可以观察到正在运行的进程，管理器显示了进程的具体运行状况，如图 5.1 所示。



进程名称	PID	CPU	CPU 时间	内存使用
System 2.61a Pro ...	0	98	5:40:11	16 K
System	0	00	0:01:45	272 K
csrss.exe	128	01	0:02:52	7,360 K
SMSS.exe	156	00	0:00:08	368 K
csrss.exe	184	00	0:01:13	8,892 K
WINGUARD.exe	204	00	0:00:07	412 K
services.exe	232	00	0:00:13	14,288 K
LSASS.exe	244	00	0:00:00	5,360 K
svchost.exe	404	00	0:00:00	3,532 K
SPoolSVR.exe	440	00	0:00:01	4,088 K
defwatch.exe	464	00	0:00:01	1,160 K
svchost.exe	504	00	0:00:00	10,612 K
lsassr.exe	532	00	0:00:00	2,276 K
adv.exe	560	00	0:00:00	2,972 K
explorer.exe	704	00	0:00:00	2,688 K
REGSVR.exe	724	00	0:00:00	3,764 K
notepad.exe	756	00	0:00:00	1,884 K
regedit.exe	824	00	0:00:00	876 K
notepad.exe	888	00	0:00:00	1,884 K

图 5.1 Windows NT 任务管理器的进程项

5.1 线程简介

对一个 Windows 程序员来说，线程提供了非常大的好处。可以在应用程序中的任何地方创建多个附属线程，它们在后台进行各种类型的处理。例如：在一个电子表格程序中计算单元格，或是脱机打印 Word 文档。即使后台正在处理许多工作，也不会影响前台的用户界面。

5.1.1 进程和线程

在深入了解线程前，要明确进程的概念。

一个进程通常为程序的一个实例，也就是调入内存准备执行的程序。在 Win32 中，进程占据 4GB 的地址空间。与它们在 MS-DOS 和 16 位 Windows 操作系统中不同，Win32 进程是没有活力的。这就是说，一个 Win32 进程并不执行什么指令，它只是占据着 4GB 的地址空间（用于存放应用程序 EXE 文件的数据以及 EXE 文件需要的任意 DLL 文件和数据）和其他系统资源（如文件、动态内存分配和同步对象）。这里的 4GB 的地址空间不是指物理地址空间，而是 Windows 的虚拟地址空间，其目的是使进程与进程互不干扰。由于每个进程都在自己的 4GB 空间内运行，它们认为自己拥有着整台计算机，而不知道其他进程同样存在于计算机中。所以每个进程在计算机中与其他进程的关系是完全独立的，它们互相覆盖的机会非常少。因此，一个程序对内存的无效访问会覆盖另一个程序或者操作系统的某些部分的情况是不可能的。每个进程都被封闭在一个安全的虚拟世界中，在其中它不能获得任何其他程序。这样的结果使进程间的通信与 Windows 3.1 中的有很大的不同。但是，这也使得计算机不会轻易崩溃，这就是使用 Win32 的好处。

通常，每个进程至少有一个线程（即主线程）在执行自己的地址空间中的代码。如果没有线程执行进程地址空间中的代码，进程也就没有继续存在的理由，于是系统将自动清除进程及其地址空间。当进程终止时，在它生命期中创建的各种资源将被清除。

1. 线程（Thread）

线程是进程内部执行的路径，是操作系统分配 CPU 时间的基本实体。每个进程都由主线程开始执行应用程序直至完成。每个进程可以包含几个线程，它们可以同时独立地执行进程的地址空间中的代码。为了做到这一点，每个线程有自己的一组 CPU 寄存器和堆栈。线程在等待分配时间片的过程，始终都保存有记录“Context”的数据结构（其中包括 CPU 寄存器、核心堆栈、线程环境块和该线程地址空间中的用户定义堆栈）。Context 结构的定义参见“Win32.hlp”文件。为了运行所有这些线程，操作系统为每个独立线程安排一些 CPU 时间，应用程序为了实现多任务并行处理，可以采用创建多个进程和在单一进程中创建多线程两种方法，但后者比前者更有效，这是因为：

- 线程的代码已经映射到了进程的地址空间，而新进程的代码还需要载入内存，所以系统创建和执行线程要比创建和执行进程快得多。

- 进程的所有线程共享进程的地址空间，并能够访问进程的全局变量，因而线程之间的通信会更便捷。

- 进程的所有线程都能使用文件和管道资源的开放句柄。

2. 创建线程

调用创建线程的 Windows API 叫做 CreateThread。实际上，线程函数可以拥有任意名称。但是它总会占用一个 32 Byte 的指针或变量作为一个参数。而且，它总是会返回一个 32 Byte 的值。线程函数的参数是 Pointer 类型，也就是说，这只是一个普通指针。正因为如此，它才能在这个变量中传递一个指向任何结构的指针。而且，还可以在这个参数中传递一个 32 Byte 的整数值。

下面是创建线程的代码:

```
hThread:=CreateThread(nil,      //安全属性
                     0,        //初使化堆栈
                     @ThreadFunc, //线程地址
                     nil,      //线程参数
                     0,        //创建 flags
                     ThreadID); //线程 ID
```

```
if hThread =0 then
```

```
    MessageBox(handle,'No Thread', nil, MB_OK);
```

这些代码只试图创建一个线程。如果出现错误, 它会弹出一个消息框, 通知用户出现了错误。

下面为 CreateThread 函数说明:

```
function CreateThread(
    lpThreadAttributes : Pointer;           //安全属性指针
    dwStackSize : DWORD;                   // 初使化线程堆栈的大小
    lpStartAddress : TFNThreadStartRoutine; //线程函数指针
    lpParameter : Pointer;                 //线程参数
    dwCreationFlags : DWORD;               //创建 flags
    var lpThreadID : DWORD);               //接受线程 ID 指针
    Thandle;                               //返回线程句柄 (Returns handle to thread)
```

第一个参数代表一系列安全属性。如果这个参数是 nil, 将会使用默认的安全属性。在 Windows 98 下, 把这个参数设置为 nil 是标准的用法, 除非想让子进程继承这个线程的句柄 (详细信息请查阅 Win32 帮助文件中的 SECURITY_ATTRIBUTES)。

如果线程的第二个参数是 0, 线程堆栈的大小和应用程序堆栈的大小是一样的。换言之, 也就是主线程和正在启动的线程具有相同大小的堆栈。如果必要的话, 堆栈自动增长, 通常可以把这个参数设置为 0。

lpStartAddress 参数是这个函数调用中最重要的部分, 它是指定在线程开始运行时调用的线程函数的名称的位置。只要在这个域中输入函数的名字, 并且在名字前加个 “@” 符号就可以了。

Windows.pas 对 TFNThreadStartRoutine 作了如下的声明:

```
TfparProc = Pointer;
```

```
TFNThreadStartRoutine = TfparProc;
```

简而言之, 这个参数的输入不受限制。但是, 如果为了调用成功, 就应输入如 ThreadFunc 这样的函数地址。

如果向函数中传递一个参数, 且要在 lpParameter 指定它, 那么, 典型的做法是, 先创建一个结构, 然后把这个地址传递到这个参数。实际上, 使用的变量不一定需要一个结构, 它也可以是一个字符串或者其他类型的变量。

dwCreationFlags 能够传递与线程有关的某些标志, 在这个域中惟一会用到的一个标志是 CREATES_SUSPENDED。如果建立一个可挂起的线程, 便是创建了线程自身, 也就创建了

它的堆栈，并且它的 TCONTEXT 结构也填入了 CPU 的值，但线程没有分配任何 CPU 时间，它可以随时启动。但是，如果不运行 ResumeThread 的话，它是不会运行的。如果愿意的话，可以通过调用 SuspendThread 来再次终止线程。可挂起线程的完整主题确实是个有关线程同步化的高级问题。就现在来说，只给这个参数赋值为 0 就可以了。

最后一个参数 lpThreadId，包含了一个由系统分配给它的惟一 ID 的可变参数。在 Windows 98 环境下，这个参数可以是 0。而且事实上，大部分时间内不会用到线程的 ID，所以可以把它习惯上设为 0。但是，在 Windows NT 的环境下，这个参数不可以为 0。如果对 Windows NT 和 Windows 98 之间的可移植感兴趣，也不应把这个参数设为 0。换句话说，如果应用程序要进入到常规产品中，应传递一个 32 位的 DWORD，而不能让这个值为 0。记住，用户并没在这个参数中传输值到 Windows 中，事实上，是在输入一个由 Windows 分配给一个值的变量。

在创建了一个未被挂起的线程之后，传递给它的函数会自动调用。换句话说，在建立了一个未被挂起的线程之后，代表这个线程的登录点的函数几乎立即就被调用了。可以看到，当以最基本标准看待它们时，线程显得一点也不复杂。当有多个在同一时刻试图访问相同数据的线程的时候，问题就比较复杂了。在这个时候，需要寻找一个途径使两个线程互相配合，这样它们才能在不产生交叉影响的前提下共同工作。

3. 线程的挂起和唤醒 (Suspend and Resume)

当线程处于挂起状态时，CPU 不会给它分配时间片，也就是说该线程将在挂起指令发出时的代码处停止，直到它被允许继续执行。挂起线程可以用于在线程执行前初始化线程状态或者在执行后修改状态，还可用于创建线程时一次性同步。

线程的挂起可调用 Windows 的 SuspendThread 函数：

Function SuspendThread(hThread : Thandle) : DWORD;

这个函数功能是将线程挂起，并且将线程被挂起的次数加 1。其中，hThread 参数是要挂起的线程的句柄。如果函数调用成功，就返回线程被挂起的次数（包括这一次）。

线程的唤醒可调用 Windows 的 ResumeThread 函数：

Function ResumeThread(hThread : Thandle) : DWORD;

这个函数功能是将线程被挂起的次数减 1，如果线程被挂起的次数减为 0，就唤醒线程。其中，hThread 参数是要挂起的线程的句柄。如果函数调用成功，就返回线程被挂起的次数。

注意：如果一个线程被挂起多次，必须相应地调用多次 ResumeThread 函数才能把线程唤醒。

5.1.2 线程的同步

在有若干线程并行运行的环境里，同步各不同线程活动的能力是非常重要的，这样可以避免对共享资源的访问冲突。事件对象是同步线程的最基本形式，它用以向其他线程发送信号以表示某一操作已经完成。例如，一个进程可能运行了两个线程。第一个线程从文件读取数据到内存缓冲区中。每当数据已被读入，第一个线程就发送信号给第二个线程并告知可以处理数据了。当第二个线程完成了对数据的处理时，它可能需要再次给第一个线程发送信号以让第一个线程能够从文件中读入下一块数据。事件可以使用 CreateEvent 函数来创建。线程

和事件在任何时候都处于两种状态之一：有信号和无信号。当线程被创建和正在运行时，它是无信号的。一旦线程终止，它就变成有信号的。线程可以通过使用 `SetEvent` 和 `ResetEvent` 函数来将事件置成有信号和无信号。

除了以上介绍的概念和函数，在通信程序中还要用到等待函数 `WaitForSingleObject()` 和重叠 I/O 操作。等待函数能使线程阻塞自身执行，而重叠 I/O 操作能使费时的操作在后台中运行。

5.1.3 线程的优先级

线程是有优先级区分的，优先级高的线程优先被执行，只有当较高优先级的线程被挂起时，较低优先级的线程才有被执行的机会。

当一个应用程序被调入内存运行时，操作系统将自动创建一个进程和一个主线程，并且进程指定优先级类。默认的优先级类是 `NORMAL_PRIORITY_CLASS` 类，不过，可以调用 Windows 的 `SetPriorityClass` 优先级函数重新指定的优先级。

进程的优先级类又称基优先级，共有下面 4 种优先级。

`IDLE_PRIORITY_CLASS` 的级别为 4；

`NORMAL_PRIORITY_CLASS` 的级别为 7（后台）9（前台）；

`HIGH_PRIORITY_CLASS` 的级别为 13；

`REALTIME_PRIORITY_CLASS` 的级别为 24。

当进程的优先级类为 `NORMAL_PRIORITY_CLASS` 时，如果进程在前台运行，其优先级是 9，如果进程在后台运行，其优先级是 7，这是为保证在前台运行的进程能更快响应用户的输入。

线程的优先级默认值就是进程的优先级类，不过可调用 Windows 的 `SetThreadPriority` 函数重新设置线程的优先级。`SetThreadPriority` 函数设置的优先级并不是优先级的绝对数，而是相对于基优先级的相对优先级。`SetThreadPriority` 函数声明为：

```
Function SetThreadPriority(
    hThread : THandle;      //线程句柄
    nPriority : Integer) :   //线程优先级
    BOOL;stdcall;          //如果函数失败，调用 GetLastError
```

其中，`hThread` 参数是线程的句柄，`nPriority` 参数是线程的相对优先级，可以为下列值。

- ☐ `THREAD_PRIORITY_ABOVE_NORMAL` 比进程的优先级大 1；
- ☐ `THREAD_PRIORITY_BELOW_NORMAL` 比进程的优先级小 1；
- ☐ `THREAD_PRIORITY_HIGHEST` 比进程的优先级大 2；
- ☐ `THREAD_PRIORITY_LOWEST` 比进程的优先级小 2；
- ☐ `THREAD_PRIORITY_NORMAL` 与进程的优先级一样。

`nPriority` 参数还可设为 `THREAD_PRIORITY_TIME_CRITICAL`，当进程的优先级类是 `IDLE_PRIORITY_CLASS`，`NORMAL_PRIORITY_CLASS`，`HIGH_PRIORITY_CLASS` 时，线程的优先级一律为 15。当进程的优先级类是 `REALTIME_PRIORITY_CLASS` 时，线程的优先级为 31。

`nPriority` 参数还可设为 `THREAD_PRIORITY_IDLE`，当进程的优先级类是

IDLE_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, HIGH_PRIORITY_CLASS 时, 线程的优先级一律为 1。当进程的优先级类是 REALTIME_PRIORITY_CLASS 时, 线程的优先级为 16。

调用 Windows 的 GetThreadPriority 函数可返回线程的相对优先级, 其声明为:

```
Function GetThreadPriority( hThread : THandle) : Integer;
```

线程的优先级不是越高越好, 要考虑整个进程中的所有线程和其他进程中的线程。

5.1.4 线程实例

要使用线程的时候, 有两个主要的任务:

(1) 建立线程。

(2) 建立一个作为“线程登录点”的函数。

在默认的情况下, 每个程序都有一个线程。从某种意义上说, 在程序一开始执行的时候, 线程就开始存在了。和项目资源中的代码不同的是, 用户所设计的线程函数或过程, 可以在整个程序中调用。

下面这个程序 MyThreadTest.pas 介绍了线程如何建立及使用:

```
unit MyThreadTest;
interface
uses
Windows, Messages, SysUtils, Classes, Graphics,
Controls, Forms, Dialogs, StdCtrls;

type
TForm1 = class(TForm)
btnUseThread: TButton;
btnNoUseThread: TButton;
procedure btnUseThreadClick(Sender: TObject);
procedure btnNoUseThreadClick(Sender: TObject);
var
Form1: TForm1;
implementation
{$R *.DFM}

//这是线程函数, 它可以放在下面程序的任何地方
function MyThreadFunc(P:pointer):Longint;stdcall;
var
i:integer;
DC:HDC;
S:string;
begin
```

```
    DC:=GetDC(Form1.Handle);
    for i:=0 to 999999 do
    begin
        S:=Inttostr(i);
        Textout(DC,100,34,Pchar(S),length(S));
    end; // for i:=0 to 100000 do 语句结束
    ReleaseDC(Form1.Handle,DC);
end;

procedure TForm1.btnUseThreadClick(Sender: TObject);
var
    //定义一个句柄
    hThread:THandle;
    ThreadID:DWord;
begin
    //创建线程，同时线程函数被调用
    hthread:=CreateThread(nil,0,@MyThreadfunc,nil,0,ThreadID);
    if hThread=0 then
        messagebox(Handle, '没有创建一个线程', nil, MB_OK);
end;

procedure TForm1.btnNoUseThreadClick(Sender: TObject);
begin
    MyThreadfunc(nil);
    //没有创建线程时，直接调用线程函数
end;
end.
```

这个程序的执行结果如图 5.2 所示。

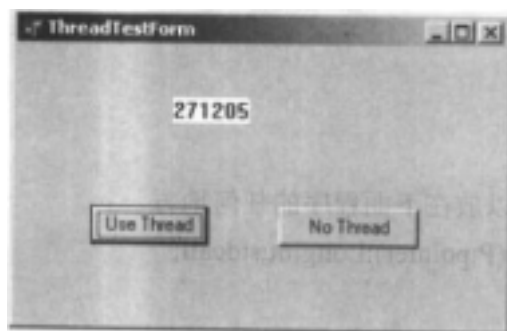


图 5.2 MyThreadTest 程序的主窗体

这个程序介绍了在使用线程及未使用线程两种情况下，运行该程序的反应。当单击“Use

Thread”按钮时，则建立一个线程，这时你可以在程序进行计算的同时，改变窗体的尺寸及移动它，因为程序执行的循环是在另一个独立的线程上运行的。当单击“NoThread”按钮时，不建立线程，会发现在程序没有计算完之前根本不能改变窗体的尺寸及移动它，因为程序处于忙的状态。

5.2 TThread 对象

了解了线程的基础知识后，下一步将讨论怎样在同一个程序中运行多个线程。在 Delphi 下进行多线程程序设计可以不去学习庞大的 Win32 API 函数，你可以利用 Delphi 标准的多线程类 TThread 来完成你的工作。Delphi 把有关线程的 API 封装在 TThread 这个 Object Pascal 的对象中。TThread 已经封装了几乎所有与线程有关的 API。下面将介绍 TThread 用法。

5.2.1 TThread 对象

TThread 对象工作的原理实际上是运用同步机制。所谓同步机制，就是事件驱动机制，意思是让线程平时处于“休眠”状态，除非发生了某个事件触发它。当某个线程要访问 VCL 时，它就调用 TThread 对象的 Synchronize 通知主线程，让主线程去真正地访问 VCL。

下面是 Classes.pas 中对 TThread 对象的说明：

```
TThread = class
private
    FHandle: THandle;
    FThreadId: THandle;
    FTerminated: Boolean;
    FSuspended: Boolean;
    FFreeOnTerminate: Boolean;
    FFinished: Boolean;
    FReturnValue: Integer;
    FOnTerminate: TNotifyEvent;
    FMethod: TThreadMethod;
    FSynchronizeException: TObject;
    procedure CallOnTerminate;
    function GetPriority: TThreadPriority;
    procedure SetPriority(Value: TThreadPriority);
    procedure SetSuspended(Value: Boolean);
protected
    procedure DoTerminate; virtual;
    procedure Execute; virtual; abstract;
    procedure Synchronize(Method: TThreadMethod);
    property ReturnValue: Integer read FReturnValue write FReturnValue;
```



```

    property Terminated: Boolean read FTerminated;
public
    constructor Create(CreateSuspended: Boolean);
    destructor Destroy; override;
    procedure Resume;
    procedure Suspend;
    procedure Terminate;
    function WaitFor: LongWord;
    property FreeOnTerminate: Boolean read FFreeOnTerminate write FFreeOnTerminate;
    property Handle: THandle read FHandle;
    property Priority: TThreadPriority read GetPriority write SetPriority;
    property Suspended: Boolean read FSuspended write SetSuspended;
    property ThreadID: THandle read FThreadID;
    property OnTerminate: TNotifyEvent read FOnTerminate write FOnTerminate;
end;

```

1. TThread 对象的特性

FreeOnTerminate 特性：布尔型，只可在运行时使用。当线程结束时，该特性决定是由 VCL（可视构件）自动清除线程对象还是由用户负责清除。当为 True 时，自动清除。默认值为 False。

Handle 特性：线程句柄，API 中大多数线程函数需要使用该句柄。

Priority：线程的优先级特性，只能在运行时使用。TThreadPriority 枚举类型定义了优先级所有可能的值。TThreadPriority 类型的值如下所述。

- ☐ tpIdle：这是线程的最低优先级，只有当系统闲置时，该优先级的线程才被执行。
- ☐ tpLowest：该优先级比正常优先级（tpNormal）低 2 个点。
- ☐ tpNormal：正常优先级。
- ☐ tpHigher：该优先级比正常优先级高一个点。
- ☐ tpHighest：该优先级比正常优先级高 2 个点。
- ☐ tpTimeCritical：线程的最高优先级。

ReturnValue 特性：整数类型，受保护特性，该特性指示线程的执行成功与否。

Suspended 特性：布尔型，用于决定线程是否挂起，当值为 True 时，挂起线程。

Terminated 特性：布尔型，且是受保护的（Protected），只读。该特性决定线程是否停止执行，如该特性值为 True 时，线程执行结束。

ThreadID 特性：线程标识符，有些 API 线程函数使用该标识。

2. TThread 对象方法

TThread 对象的方法有很多，下面介绍常用的几种方法。

DoTerminate 方法：该过程只能由线程对象内部方法调用，用于与主 VCL 线程的同步，并产生 OnTerminate 的事件，一般说来，当线程终止时，线程会自动调用该方法，不需要写代码。

Execute 过程：该方法开始一个线程的执行，用户必须在线程类中，重写该过程。Execute 方法返回时，终止线程的执行、释放线程所占资源，并调用 OnTerminate 事件过程。Execute 方法必须周期性地检测 Terminate 特性，该特性一旦为 True，Execute 必须立即返回。

Resume 过程：该方法恢复一个被挂起的线程的执行。

Suspend 过程：该方法挂起一个线程的执行，它和 Resume 方法是配对的。

Synchronize 过程：该方法是 TThread 对象的精髓。Delphi 中的各种 VCL 构件都是临界资源，只能由主线程使用，其他的线程要使用这些 VCL 构件，必须使用 Synchronize 方法以避免多线程与 VCL 构件的冲突。为此 Synchronize 过程采取的策略为用 Method 参数指定一个方法，这个方法是用来访问 VCL 的，但线程自己并不直接调用这个方法，而是通知主线程，让主线程去调用这个方法。在同一时刻，主线程只能收到一个通知，因此能保证不会有几个线程同时访问 VCL。

Terminate 过程：该方法设置 Terminated 特性为 True，并终止线程的执行。

WaitFor 函数：该方法等待线程执行的终止，然后返回 ReturnValue 特性的值，因此，在调用该函数后，必须确保线程的退出。另外，如果线程使用了 Synchronize 方法，则不要在主线程中调用 WaitFor，因为这样一来易引起死锁，或导致 Ethread 例外的发生。

OnTerminate 过程：该过程是 OnTerminate 事件的驱动程序。OnTerminate 事件发生在线程的 Execute 方法已经返回，TThread 结束线程之前，该事件驱动只可在主线程使用，可以调用各种 VCL 方法和特性。

3. TThread 的用法简介

TThread 是直接从 TObject 继承下来的，是一个抽象 (Abstract) 类 (一个带有虚拟方法的类)。它和其他的 Delphi 构件和对象不同，它不是组件，不能在程序中直接使用。这意味着，不能创建 TThread 的实例，而只能创建其派生类的实例，并对其中的某些成员函数进行重写以覆盖 (Override) TThread 对象中的方法，完成自己的功能。在 Delphi 中可以直接书写代码创造自己的线程类，也可以和 Delphi 的线程生成器来创建原始的公用代码，而后再在此基础上修改。

可以选择 Delphi 的主菜单中的“File→New”命令，然后在 New Items 对话框中双击 Thread Object 图标，如图 5.3 所示。当双击 Thread Object 图标后，将出现一个对话框，它会提示输入线程对象的名称。



图 5.3 在 New Items 对话框中的 Thread Object 项

例如, 输入 SimpleThread。Delphi 会自动创建一个包括新创建的线程对象的单元, 如下所示:

```
Type
TSimpleThread = class(TThread)
private
protected
procedure Execute ; override;
end;
```

TThread 的派生类中惟一必须覆盖的方法是 Execute。假设用户要在 TSimpleThread 中进行复杂的计算, 可以如下定义 Execute:

```
procedure TSimpleThread.Execute;
var Intx,Inty: integer;
begin
  for Intx:=2 to 9999999 do
    inc(Inty, Round(Abs(Cos(Sqrt (Intx)))));
end;
```

当然, 本段代码只是演示一个长循环。

现在, 可以通过调用线程对象的 Create 使上述代码执行。在这个例子中, 在主窗体上放置一个按钮, 单击此按钮就会调用 Create。注意, 不要忘记在主窗体单元的 uses 子句中包含 TTestThread 单元, 否则会导致编译错误。

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  SimpleThread.Create(False);
end;
```

如果运行此程序并单击按钮, 就会执行上述的长循环, 但同时仍然可以操纵窗口, 做各种其他操作。

5.2.2 TThread 实例

下面例子说明线程优先级的应用, 该程序同时使用了多个线程, 允许用户使用 TrackBar 组件来改变优先级。本例子用 ProgressBar (进程条) 来显示线程的执行情况。

本例子的详细代码及说明如下所述。

首先新建一个单元, 保存为 mythread.pas。在此单元中, 创建一个 TMyThread 类, 它从 TThread 类中继承而来, 所以创建这个类的一个实例时, 也就是创建了一个新的线程。单元中构造函数 CreateIt 用于将线程的优先级与进程条联系起来。当线程对象初始化后, 将 Suspend 的值设为 False, 这是为了唤醒 Thread, 让它立即执行。

mythread 单元体的程序如下:

```
unit mythread;
interface
uses
```

Classes, comctrls;

type

TMyThread = class(TThread)

private

PB : TProgressBar;

procedure InitProgressBar; // 初使化 ProgressBar

procedure UpdateProgressBar; // 修改 ProgressBar

protected

procedure Execute; override; // 主要 thread 执行

published

constructor CreateIt(PriorityLevel: cardinal; ProgBar : TProgressBar);

destructor Destroy; override;

end;

implementation

uses

windows, thread;

constructor TMyThread.CreateIt(PriorityLevel: cardinal; ProgBar : TProgressBar);

begin

inherited Create(true);

Priority := TThreadPriority(PriorityLevel); // 设置优先级别

FreeOnTerminate := true; // 当结束时, Thread 释放自己

PB := ProgBar;

Synchronize(InitProgressBar); //初使化 ProgressBar 后, 用 Synchronize 方法

Suspended := false; // 继续这个 thread

end;

destructor TMyThread.Destroy;

begin

//发送一个 Message 到主窗体, 告知何时执行 Thread 和哪个 Thread 执行

PostMessage(form1.Handle,wm_ThreadDoneMsg,self.ThreadID,0);

inherited destroy;

end;

// 执行线程

procedure TMyThread.Execute;

var

```

i : cardinal;
begin
  i := 1;
  while ((Terminated = false) and (i < 100000)) do
  begin
    Synchronize(UpdateProgressBar);
    //修改 ProgressBar, 用 Synchronize, 因为 ProgressBar 在另一个线程中
    Inc(i);
    //如果 Terminated 为 True, 循环将退出来, 正在执行的线程将结束
  end;
end;
procedure TMyThread.InitProgressBar; // 初使化 ProgressBar
begin
  PB.Min := 1;
  PB.Max := 100000;
  PB.Step := 1; // 设置 Step 为 1
  PB.Position := 1; //设置开始的位置
end;

procedure TMyThread.UpdateProgressBar; //修改 ProgressBar
begin
  PB.StepIt; //把进程条的 Step 加 1
end;
end.

```

接着创建主程序 TThread 的窗体, 主程序将调用 mythread 单元。TThread 程序的主窗体显示如图 5.4 所示。在窗体中放置如图 5.4 的控件。单元中声明了一个 WM_ThreadDoneMsg 消息常量, 当一个线程的 Execute 方法执行完后, 发出这个消息, 当消息句柄收到这个消息后, 进行消息处理, 即把进程条的 Step 加 1。当用户修改 TrackBar 的刻度尺, 将触发 TrackBar 的 OnChange 事件, 此事件的处理是根据用户的要求, 重设线程的优先级。

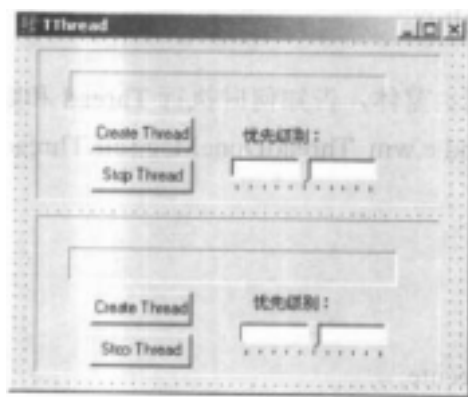


图 5.4 Tthread 程序的主窗体

具体代码如下:

```
unit thread;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ComCtrls, ExtCtrls, mythread;
const
  WM_ThreadDoneMsg = WM_User + 8;

type
  TForm1 = class(TForm)
    ProgressBar1: TProgressBar;
    ProgressBar2: TProgressBar;
    Button1: TButton;
    Button2: TButton;
    TrackBar1: TTrackBar;
    TrackBar2: TTrackBar;
    Label1: TLabel;
    Label2: TLabel;
    Panel1: TPanel;
    Panel2: TPanel;
    Button3: TButton;
    Button4: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure TrackBar1Change(Sender: TObject);
    procedure TrackBar2Change(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    { Private declarations }
    MyThread1: TMyThread;           // thread 1
    MyThread2: TMyThread;           // thread 2
    Thread1Active: boolean;         // 用于测试 Thread1 是否在运行
    Thread2Active: boolean;         // 用于测试 Thread2 是否在运行
    procedure ThreadDone(var AMessage: TMessage); message WM_ThreadDoneMsg;
    // 当一个线程的 Execute 方法执行完后, 发出这个消息
```

```
public
    { Public declarations }
end;

var
    Form1: TForm1;
implementation
    {$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject); // 创建 Thread 1
begin
    if (MyThread1 = nil) or (Thread1Active = false) then //确认 thread 没运行
    begin
        MyThread1 := TMyThread.CreateIt(TrackBar1.Position, ProgressBar1);
        Thread1Active := true;
    end
    else
        ShowMessage('Thread still executing');
end;

procedure TForm1.Button2Click(Sender: TObject); // 创建 Thread 2
begin
    if (MyThread2 = nil) or (Thread2Active = false) then // 确认 thread 没运行
    begin
        MyThread2 := TMyThread.CreateIt(TrackBar2.Position, ProgressBar2);
        Thread2Active := true;
    end
    else
        ShowMessage('Thread still executing');
end;

procedure TForm1.Button3Click(Sender: TObject); // 结束 Thread 1
begin
    if (MyThread1 <> nil) and (Thread1Active = true) then // 检查 Thread 是否正在运行
        MyThread1.Terminate
    else
        ShowMessage('Thread not started');
end;
```

```

procedure TForm1.Button4Click(Sender: TObject); // 结束 Thread 2
begin
    if (MyThread2 <> nil) and (Thread2Active = true) then //检查 Thread 是否正在运行
        MyThread2.Terminate
    else
        ShowMessage('Thread not started');
end;

procedure TForm1.ThreadDone(var AMessage: TMessage);
begin
    if ((MyThread1 <> nil) and (MyThread1.ThreadID = cardinal(AMessage.WParam))) then
        begin
            Thread1Active := false;
        end;
    if ((MyThread2 <> nil) and (MyThread2.ThreadID = cardinal(AMessage.WParam))) then
        begin
            Thread2Active := false;
        end;
end;

procedure TForm1.FormCreate(Sender: TObject); // 初使化
begin
    Thread1Active := false;
    Thread2Active := false;
end;

procedure TForm1.TrackBar1Change(Sender: TObject); // 设置 Thread 1 优先级
begin
    if (MyThread1 <> nil) and (Thread1Active = true) then
        MyThread1.priority := TThreadPriority(TrackBar1.Position);
end;

procedure TForm1.TrackBar2Change(Sender: TObject); // 设置 Thread 2 优先级
begin
    if (MyThread2 <> nil) and (Thread2Active = true) then
        MyThread2.priority := TThreadPriority(TrackBar2.Position);
end;

procedure TForm1.FormDestroy(Sender: TObject); //结束运行的线程

```



```
begin
  if (MyThread1 <> nil) and (Thread1Active = true) then
  begin
    MyThread1.Terminate;
    MyThread1.WaitFor; // 等待 Thread 结束
  end;
  if (MyThread2 <> nil) and (Thread2Active = true) then
  begin
    MyThread2.Terminate;
    MyThread2.WaitFor;
  end;
end;
```

本章小结

本章主要是简单地讲述了后面章节所要用到的线程技术。首先介绍了进程和线程的基本概念、线程的创建、线程的挂起和唤醒、线程的优先级以及线程的同步，然后介绍了 Delphi VCL 的 TThread 对象、TThread 特性和方法等内容。后面的章节将多次用到本章的知识。

第 6 章 Windows API 通信编程

本章主要内容：

- Windows 通信结构
- 串口通信 API 函数介绍
- TAPI 主要函数和基于 TAPI 应用的步骤介绍
- 示例程序和分析

计算机通信资源（也叫通信设备）指的是能够提供单一的双向同步数据流的物理或逻辑设备，如串口、并口、传真机和调制解调器等。Windows 操作系统为所有通信设备提供了一套驱动程序库以便应用程序访问它们，例如在 Windows 98/NT 中的串行设备驱动程序是“comm.drv”，这些驱动程序主要完成对信号电平的判读、串行设备芯片（如 INS8250）与 CPU 之间信号和数据的传递。一切基于串口的计算机通信操作和通信编程都是在此基础上进行的。

本章介绍 Windows API 通信编程的基础知识，并给出了详细的例子，分析如何使用 Windows API 和 TAPI 函数实现通信。

6.1 串口通信 API 函数

Win32 通信编程接口（API）用来以交互的流模式发送或者接收数据。在设置通信配置和发送错误敏感（Error Sensitive）、无时间限制（Non Time Critical）的数据时，该接口尤其有用。比如，可以使用此编程接口编辑共享文档、玩多人游戏或模仿交互式的情境等。WAVE API 用来设置和实时发送或者接收可容错的（Error Tolerant）、时间敏感的（Time Sensitive）音频数据（Audio Data）。MAPI 可用来发送和接收文件、传真、E-mail 或者处理任何非交互的、基于消息的传输。TAPI 用来连接、控制和断开电话呼叫。

本节首先介绍 Windows 98 和 Windows 3.x 通信结构，再介绍串口通信 API 函数，然后给出例子展示 Windows API 函数的功能。

6.1.1 Windows 98 和 Windows 3.x 通信结构

Windows 98 与 Windows 3.x 在通信机制上有很大差异。Windows 98 采用的是 32 位通信子系统，Windows 3.x 采用的是 16 位通信子系统。Win32 和 Win16 通信结构如图 6.1 所示。

Win16 通信结构提供的 comm.drv 既能作为应用与设备的中间接口，又能作为通信端口的驱动程序。由于要在不同模式间进行转换，因此大大降低了处理高速通信时的可靠性。软

硬件开发商为了追求超强通信功能，不得不去重写驱动程序以取代 comm.drv（实质上相当于改写通信子系统），这使得系统的兼容性或稳定性大受影响。

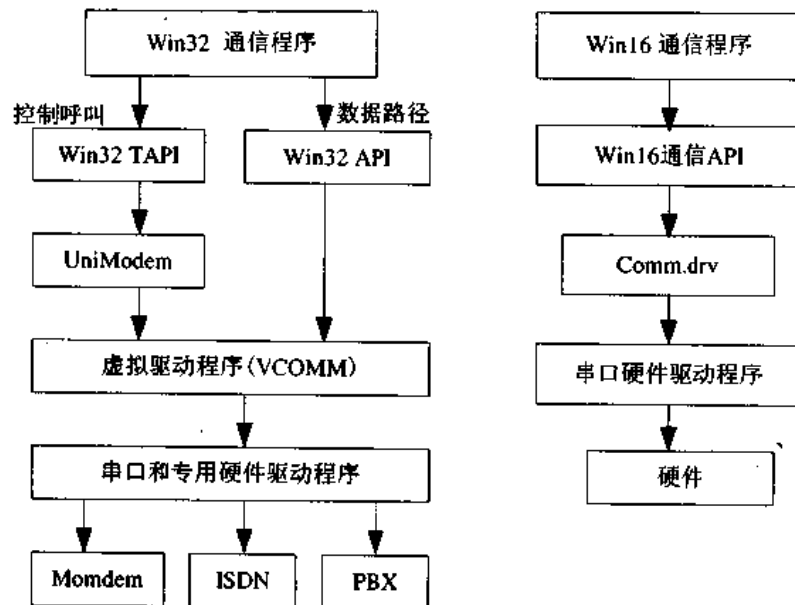


图 6.1 Win32 和 Win16 通信结构

与此相反，Win32 提供一个模块化的、32 位的、保护模式的通信子系统。该系统功能强大，针对 Win16 结构上反映出的问题，用灵活的通信子系统对原有的通信结构进行了彻底改造。Win16 的单块结构被代之以多块结构。

Win32 API 和 Win32 TAPI 这两块共同形成应用接口。其中，由 Win32 API 负责设置 Modem 和传输数据，Win32 TAPI 用来控制连接。TAPI 在本章的后面会详细介绍。

位于 Win32 TAPI 之下的 UniModem 的功能是提供调制解调器服务及语音/数据的切换操作。UniModem 用 Modem 开发商开发的微型驱动程序自动处理模糊的 AT 命令来实现拨号，应答及设置 Modem 和类似的设备。微型驱动程序包括特定硬件命令，将归并在 Windows Registry（注册数据库）中。当一个通信程序准备连接一个 Modem 时，UniModem 在 Registry 中读到相应的 Modem 命令并把它们传给 VCOMM，UniModem 不仅是一个 VCOMM 设备驱动程序，还作为 TAPI 服务提供商。

VCOMM 位于 UniModem 之下，提供一条从通信应用程序到硬件之间的保护模式代码通路，实现可靠的高速数据传输。VCOMM 是一个 VxD（虚拟设备驱动器），它会在 Windows 98/NT 启动时装入，对于其他开发商开发的、用于与他们自己的适配器进行通信的专用硬件端口驱动程序而言，VCOMM 像一个守门员，只有应用程序需要这些驱动程序时，VCOMM 才装入它们，这样就节约了资源。最后，VCOMM 还利用了 Windows 95 的即插即用服务来帮助用户安装和设置新的通信设备。

端口驱动程序是动态装入的虚拟设备驱动程序，它用于写 I/O 端口通信。

Win32 有了这种分层结构，使应用程序可在通信子系统之外，通过 API 实现许多想要的功能。

6.1.2 串口通信 API 函数介绍

Windows 系统通信一般都以 WOSA (Windows Open Services Architecture, 即 Windows 开放式服务体系) 模型为基础, 在此模型中位于上层的应用程序通过调用各种通信 API (Application Programming Interfaces, 即应用程序接口) 与位于下层的设备驱动程序进行数据交换。

Windows 操作系统的机制禁止应用程序直接访问计算机硬件, 但它提供了丰富的应用程序接口, 可以支持大多数的硬件和协议, 并隐藏了许多低层的处理, 为编程人员编制通信程序提供了方便, 免除了编程人员对有关硬件的调试麻烦。

下面介绍 Windows 环境下编写串口通信程序所需的 API 函数知识, 详细说明请参阅 win32.hlp 文件。

1. DCB 数据结构

DCB 结构为串行通信设备定义了一些控制设置。

(1) 定义

```
typedef struct _DCB {          // dcb
    DWORD DCBlength;           // DCB 结构大小
    DWORD BaudRate;             // 波特率
    DWORD fBinary: 1;           // 二进制模式
    DWORD fParity: 1;           // 进行奇偶校验
    DWORD fOutxCtsFlow: 1;      // 使 CTS 信号进行输出流量控制
    DWORD fOutxDsrFlow: 1;      // 使 DSR 信号进行输出流量控制
    DWORD fDtrControl: 2;       // DTR 流量控制
    DWORD fDsrSensitivity: 1;   // DSR 敏感度
    DWORD fTXContinueOnXoff: 1;  // XOFF 后是否继续发送
    DWORD fOutX: 1;             // 使得输出 XON/XOFF 有效
    DWORD fInX: 1;              // 使得输入 XON/XOFF 有效
    DWORD fErrorChar: 1;        // 允许奇偶错误替换
    DWORD fNull: 1;             // 允许删除 Null
    DWORD fRtsControl: 2;       // RTS 流量控制
    DWORD fAbortOnError: 1;     // 出错时是否终止读写操作
    DWORD fDummy2: 17;          // 保留
    WORD wReserved;             // 当前未用, 必须置为 0
    WORD XonLim;                // XON 阈值
    WORD XoffLim;               // XOFF 阈值
    BYTE ByteSize;              // 字符位数, 4~8
    BYTE Parity;                // 奇偶校验位, 0~4 分别为 no、odd、even、mark、space
    BYTE StopBits;              // 0, 1, 2 分别为 1, 1.5, 2
    char XonChar;               // XON 字符
    char XoffChar;              // XOFF 字符
};
```

```

    char ErrorChar;           // 奇偶错误替代字符
    char EofChar;             // 结束字符
    char EvtChar;             // 事件字符
    WORD wReserved1;          // 保留, 未用
} DCB;

```

(2) 说明

DCBlength: 用 byte 作单位来指定 DCB 的长度。

BaudRate (波特率): 指定通信设备的传输速率。这个成员可以是实际波特率值或者下面的波特率序数值:

```

    CBR_110    CBR_19200  CBR_300    CBR_38400  CBR_600  CBR_56000
    CBR_1200   CBR_57600  CBR_2400   CBR_115200 CBR_4800  CBR_128000
    CBR_9600   CBR_256000 CBR_14400

```

fBinary: 指定二进制使能模式。Win32 API 不支持非二进制模式传输, 因此这个成员必须是 TRUE。若用 FALSE 则将不工作。在 Windows 3.1 下, 若此成员设为 FALSE, 非二进制模式使能, 通过设定下面的 EofChar 成员字符作为输入的结束字符。

fParity: 指定奇偶校验使能。若成员为 TRUE, 将进行奇偶校验检查并进行错误报告。

fOutxCtsFlow: 指定 CTS (Clear-To-Send) 信号是否被监视并作为输出流量控制信号。若此成员设为 TRUE 且 CTS 信号关闭, 则输出被挂起, 直到 CTS 重新发出。

fOutxDsrFlow: 指定 DSR (Data-Set-Ready) 信号是否被监视并作为输出流量控制信号。若此成员设为 TRUE 且 DSR 信号关闭, 则输出被挂起, 直到 DSR 重新发出。

fDtrControl: 指定 DTR (Data-Terminal-Ready) 信号作为流量控制。此成员可以是表 6.2 中列出的值。

表 6.1 DTR 流量控制

值	含义
DTR_CONTROL_DISABLE	当串口设备打开时, DTR 信号线始终设为禁止
DTR_CONTROL_ENABLE	当串口设备打开时, DTR 信号线设为使能并始终打开
DTR_CONTROL_HANDSHAKE	DTR 握手使能, 若握手使能, 可调用 EscapeCommFunction 函数来释放 DTR 线上的错误

fDsrSensitivity: 指定通信设备对 DSR 信号状态是否敏感。若此成员设为 TRUE, 除非 DSR 信号线上的电平为高, 否则接收到的字节将被忽略。

fTXContinueOnXoff: 指定当输入缓冲区满且收到 XoffChar 字符时传输是否停止。若此成员设为 TRUE, 在输入缓冲区小于 XoffLim 定义的字节数传输继续, 发完 XoffChar 字节后停止接收字节。若此成员设为 FALSE, 输入缓冲区直到多于 XonLim 定义的字节数为空时, 传输才会继续且驱动程序发送过 XonChar 字符后接收过程才恢复。

fOutX: 指定在发送期间是否使用 XON/XOFF 流量控制。若此成员设为 TRUE, 当收到 XoffChar 字符时停止发送, 收到 XonChar 字符时开始发送。

fInX: 指定接收期间是否使用 XON/XOFF 流量控制。若此成员设为 TRUE, 当输入缓冲

区中存有数据的空间达到 XoffLim 定义的字节数时, XoffChar 被发送; 当输入缓冲区中可用的空间达到 XonLim 定义的字节数时, XonChar 被发送。

fErrorChar: 指定当收到的字节发生奇偶校验错误时是否用 ErrorChar 成员定义的字符代替。若此成员设为 TRUE 且 fParity 成员设为 TRUE, 将会发生替换。

fNull: 指定 null 字节是否被丢弃。若此成员设为 TRUE, 接收到的 null 字节被放弃。

fRtsControl: 指定 RTS (request-to-send) 流量控制。若此值为 0, 缺省值为 RTS_CONTROL_HANDSHAKE。此成员可用值在表 6.2 中列出。

表 6.2 RTS 流量控制

值	含义
RTS_CONTROL_DISABLE	当设备打开时, RTS 线始终设为禁止
RTS_CONTROL_ENABLE	当设备打开时, RTS 线始终设为使能并始终打开
RTS_CONTROL_HANDSHAKE	RTS 握手使能。当输入缓冲区字符小于缓冲区总数的 1/2 时, 驱动程序使 RTS 信号为高, 超过 3/4 时使 RTS 信号为低。若握手使能, 可调用 EscapeCommFunction 函数来释放 DTR 线上的错误
RTS_CONTROL_TOGGLE	指定当发送数据时, RTS 信号线为高。缓冲区数据发送完, RTS 信号线为低

fAbortOnError: 指定当错误发生时, 读写操作是否终止。若此成员设为 TRUE, 当错误发生时所有的读写操作将终止, 驱动程序直到应用程序识别错误并调用 ClearCommError 函数才进行更深层次的通信。

fDummy2: 保留, 未用。

WReserved: 未用, 必须设为 0。

XonLim: 指定发送 XON 字符前输入缓冲区允许的最小字节数。

XoffLim: 指定发送 XOFF 字符前输入缓冲区允许的最大字节数。最大允许接收字符数用定义的全部输入缓冲区字节数减去此值得到。

ByteSize: 指定每字节发送接收的位数。

Parity: 指定奇偶校验方法。此成员可用值在表 6.3 列出。

表 6.3 Parity 的值

值	含义
EVENPARITY	偶校验
NOPARITY	无校验
MARKPARITY	标记校验
ODDPARITY	奇校验

StopBits: 指定停止位的位数。此成员可用值在表 6.4 列出。

表 6.4

StopBits 的值

值	含义
ONESTOPBIT	1 位停止位
ONE5STOPBITS	1.5 位停止位
TWOSTOPBITS	2 位停止位

XonChar: 指定发送接收的 XON 字符。

XoffChar: 指定发送接收的 XOFF 字符。

ErrorChar: 指定当接收的奇偶校验错误时被代替的接收字符。

EofChar: 指定某字符作为数据结束标记。

EvtChar: 指定某字符作为事件标记。

wReserved1: 保留, 未用。

注意: 当 DCB 结构配置被 8250 使用时, ByteSize 和 StopBits 必须按下列方式限定:

数据位必须是 5 到 8 位, 5 位数据位配合 2 位停止位, 6、7 或 8 位数据位配合 1.5 位数据位。

2. 串口通信 API 函数

下面介绍重要的串口通信 API 函数, 详细说明请见 Win32 API 帮助的“Communication Functions”一节。

❑ **CreateFile**——CreateFile 函数创建或打开下列对象并且返回一个句柄, 以便被其他对象访问, 即功能为打开串行口。

```
HANDLE CreateFile(
    //文件名指针
    LPCTSTR lpFileName,
    //访问(读写)模式
    DWORD dwDesiredAccess,
    DWORD dwShareMode, //共享模式
    //安全属性指针
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDistribution, //创建方法
    DWORD dwFlagsAndAttributes, //文件属性
    HANDLE hTemplateFile
);
```

❑ **BuildCommDCB**——BuildCommDCB 函数用指定的设备控制字符串(device-control string)以填充特定的 DCB 结构。设备控制字符串用相应的模式控制命令得到。

```
BOOL BuildCommDCB(
    //设备控制字符串指针
    LPCTSTR lpDef,
```

//设备控制块指针

LPDCB lpDCB

);

□ **BuildCommDCBAndTimeouts**——**BuildCommDCBAndTimeouts** 函数把一个设备定义字符串 (device definition string) 发送给设备控制块。此函数能设置时间溢出 (time out) 值、时间未溢出 (no time-outs) 值, 这些设置都基于设备定义字符串内容的改变。

□ **ClearCommBreak**——**ClearCommBreak** 函数对指定的通信设备回复字符发送, 并把发送线置于 nonbreak 状态。

BOOL ClearCommBreak(

HANDLE hFile

//通信设备句柄, 是 CreateFile 函数返回的句柄

);

□ **ClearCommError**——**ClearCommError** 函数回复通信错误信息并报告当前的通信设备状态。当通信错误发生时调用此函数, 它会清除附加的 I/O 操作的设备错误标志。

BOOL ClearCommError(

//通信设备句柄

HANDLE hFile,

//接收错误代码的变量指针

LPDWORD lpErrors,

LPCOMSTAT lpStat // 通信状态缓冲器指针

);

lpErrors: 32 位变量指针, 其可能的值在表 6.5 中列出。

表 6.5 通信错误码

值	含义
CE_BREAK	硬件检测到一个中断状态
CE_DNS	Windows 95/98: 并行口未被选择
CE_FRAME	硬件检测到一个帧错误
CE_IOE	设备通信时发生一个 I/O 错误
CE_MODE	请求模式未被支持, 或者 hFile 参数无效。如果这个值被指定, 它就是仅有的合法错误
CE_OOP	Windows 95/98: 并行设备通知无纸
CE_OVERRUN	一个字符缓冲区侵占发生, 下一个字符被丢失
CE_PTO	Windows 95/98, 并行设备延时溢出
CE_RXOVER	输入缓冲区溢出发生。或者输入缓冲区无空间, 或者在收到 End-Of-File (EOF) 字符之后又接收到字符
CE_RXPARITY	硬件检测到奇偶校验错误
CE_TXFULL	应用程序试图发送一个字符, 但是输出缓冲区已满

lpStat: 一个 COMSTAT 结构指针, 返回设备的状态信息。

❑ CommConfigDialog——CommConfigDialog 函数显示驱动程序提供的配置对话框。

❑ DeviceIoControl——DeviceIoControl 函数把控制码直接发送给指定的设备驱动程序, 引起相应的设备执行特定的操作。

❑ EscapeCommFunction——EscapeCommFunction 函数直接让指定的通信设备执行一个扩展功能。

```

BOOL EscapeCommFunction(
    HANDLE hFile,      //通信设备句柄
    DWORD dwFunc       //扩展函数

```

```
);
```

dwFunc: 指定的扩展函数代码, 可以为下列值:

CLRDRTR 清除 DTR (data-terminal-ready) 信号

CLRRTS 清除 RTS (request-to-send) 信号

SETDRTR 发送 DTR (data-terminal-ready) 信号

SETRTS 若一个 XOFF 字符接收到将引起发送

SETXON 若一个 XON 字符已接收到将引起发送

SETBREAK 中止字符发送, 并且置发送线为 break 状态, 直到 ClearCommBreak 函数被调用

CLRBREAK 为恢复发送字符, 并且置发送线为 nobreak 状态。CLRBREAK 扩展功能和 ClearCommBreak 函数功能一样

❑ GetCommState——GetCommState 函数用指定通信设备的当前控制设置填充设备控制块 (DCB)。

❑ GetCommTimeouts——GetCommTimeouts 函数对指定的通信设备返回一个时间溢出 (time out) 参数值。

❑ SetCommMask——SetCommMask 函数指定一系列事件监视通信设备。

❑ SetCommState——SetCommState 函数用相应的设备控制块配置通信设备。此函数重新初始化所有硬件和控制设置, 但是它不清空输入或输出队列。

❑ SetCommTimeouts——SetCommTimeouts 函数对基于通信设备的所有读写操作设置时间溢出参数。

❑ SetupComm——SetupComm 函数对指定的通信设备初始化相关参数。

❑ ReadFile——ReadFile 函数功能为读串口操作

```

BOOL ReadFile(
    HANDLE hFile,          //被读文件的句柄
    LPVOID lpBuffer,      //指向接收数据的缓冲区
    DWORD nNumberOfBytesToRead, //要从文件或通信资源中读取的字节数
    LPDWORD lpNumberOfBytesRead, //实际从文件或通信资源中读取的字节数
    LPOVERLAPPED lpOverlapped //可设置成空

```

```
);
```

hFile: 被读文件的句柄, 在通信应用程序中它指向由 CreateFile 打开资源的句柄,

也就是 CreateFile 的返回值。

lpBuffer: 指向接收数据的缓冲区, 该缓冲区在 Delphi 中必须被定义成如下格式:

lpReadBuffer: Array [0..1024] of char.

nNumberOfBytesToRead: 指明要从文件或通信资源中读的字节数。

lpNumberOfBytesRead: 指明实际从文件或通信资源中读的字节数。

lpOverlapped: 在通信程序中可设置成空。

❑ WriteFile——WriteFile 函数功能为写串口操作

WriteFile 与 ReadFile 的参数设置类似, 这里不再叙述。在使用 WriteFile 和 ReadFile 时, 通信设备句柄必须对该设备具有 GENERIC_READ 或 GENERIC_WRITE 访问权。

❑ CloseHandle——CloseHandle 函数功能为关闭串行口

BOOL CloseHandle(

HANDLE hObject // 指向通信资源的句柄

);

hObject: 指向通信资源的句柄。

6.1.3 示例程序和分析

在 Win32 操作系统中, 程序的执行是靠事件驱动的, 对于每个事件 (如鼠标的移动、键盘的按键、时钟的触发、应用程序的退出等), 操作系统都产生相应的消息, 并把这些消息按照一定的规则, 在 Windows 的消息队列中排队等待发送, 或者直接发送到目标窗体的消息队列中, 当某个消息被处理时, 目标窗体的窗体过程就指引控制转向相应的消息处理程序。

Windows 3.x 操作系统定义了 WM_COMMNOTIFY 消息, 当串口中断 (INT4、INT3) 被触发时发出, 串口通信程序可以在 WM_COMMNOTIFY 的消息处理程序中读取数据和信号以及再做进一步处理。在 Windows 98/NT 中, 原来 Windows 3.x 的 WM_COMMNOTIFY 消息已被取消, 操作系统为每个通信设备开辟了用户可定义大小的读/写缓冲区, 数据进出通信口均由操作系统后台完成, 应用程序只需对读/写缓冲区操作即可。

在 Win32 中进行串行通信除了解基本的通信操作函数外, 还要掌握多线程编程。串行通信需要利用多线程技术来实现。其主要的处理逻辑可以表述如下: 进程一开始先由主线程做一些必要的初始化工作, 然后主线程根据需要在适当时候建立通信监视线程, 用来监视通信口。当指定的串行口事件发生, 向主线程发送 WM_COMMNOTIFY 消息 (由于 Windows 98 取消了 WM_COMMNOTIFY 消息, 因此必须自己创建), 主线程对其进行处理。若不需要 WM_COMMNOTIFY 消息, 则主线程终止通信监视线程。

多线程同时执行, 将会引起共享资源的冲突。为避免冲突, 就要用同步多线程对共享资源进行访问。Windows 98 提供了许多保持线程同步的方法, 笔者采用创建事件对象来保持线程同步。通过 CreateEvent 创建事件对象, 使用 SetEvent 或 PluseEvent 函数将事件对象设置成信号同步。在应用程序中, 利用 WaitSingleObject 函数等待同步的触发, 等到指定的事件被其他线程设置为有信号时, 才继续向下执行程序。

下面讲一个实例程序, 功能为利用一条串行数据线连接在两台计算机 COM2 之间就可以进行文本文件传输。希望对于使用 Windows 提供的 API 标准函数编写通信程序有更深入的了解。

Delphi 的强大功能和支持多线程的面向对象编程技术能够简单方便地实现串行通信。它

通过调用外部的 API 函数来实现，主要步骤如下：

首先，用 `CreateFile` 函数打开指定的通信端口，以确定本应用程序对此串行的占有权，并封锁其他应用程序对此串口的操作。设置参数中访问类型为 `GENERIC_READ` 或 `GENERIC_WRITE`、共享模式为 0、创建标志为 `OPEN_EXISTING`、模板句柄为 `NULL`，因为程序使用异步 I/O 读写串口，所以属性标志置为 `FILE_FLAG_OVERLAPPED`，且在程序中为读写串口的函数定义了 `OVERLAPPED` 结构的变量 `Os`、`Read_Os`、`Write_Os`。当 `FILE_FLAG_OVERLAPPED` 被指定，操作系统不能维护文件指针。文件位置必须作为 `lpOVERLAPPED` 参数的一部分传递(请看 `OVERLAPPED` 结构)，以便被 `ReadFile` 和 `WriteFile` 函数使用。

然后，通过 `SetupComm` 函数给通信的输入输出队列分配一定大小的内存缓冲区，接着通过调用 `GetCommState` 函数和 `SetCommState` 函数配置串行的波特率、数据位、校验位和停止位和流控制方式，并且可以恢复缺省值。

初始化完成后就可以利用 `ReadFile` 函数和 `WriteFile` 函数对通信端口进行读写操作了。程序界面如图 6.2 所示。



图 6.2 Commutate 程序的主窗体

程序如下：

```
unit commutate;
interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Buttons, StdCtrls, ComCtrls;
const
  WM_COMMNOTIFY = WM_USER + 1; // 通信消息
type
  TForm1 = class(TForm)
```

```

    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    OpenDialog1: TOpenDialog;
    Label1: TLabel;
    RichEdit1: TRichEdit;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
private
    { Private declarations }
    procedure WMCOMMNOTIFY(var Message :TMessage);
    message WM_COMMNOTIFY;
public
    { Public declarations }
end;
var
    Form1: TForm1;
implementation
    {$R *.DFM}
var
    hNewCommFile,Post_Event: THandle;
    Read_os : Toverlapped;
    Receive :Boolean;
    ReceiveData : Dword;

// 接收的数据送入显示区
procedure AddToMemo(Str:PChar;Len:Dword); begin
    str[Len]:=#0;
    Form1.RichEdit1.Text:=Form1.RichEdit1.Text+StrPas(str);
end;

// 通信监视线程
procedure CommWatch(Ptr:Pointer);stdcall;
var
    dwEvtMask,dwTranser : Dword;
    OK: Boolean;

```

```

    Os : Toverlapped;
begin
    Receive := True;
    FillChar(Os, SizeOf(Os), 0);
    Os.hEvent := CreateEvent(nil, True, False, nil);
    // 创建重叠读事件对象
    if Os.hEvent = nil then
    begin
        MessageBox(0, 'Os.Event Create Error !', 'Notice', MB_OK);
        exit;
    end;
    if (not SetCommMask(hNewCommFile, EV_RXCHAR)) then
    begin
        MessageBox(0, 'SetCommMask Error !', 'Notice', MB_OK);
        exit;
    end;
    while(Receive) do
    begin
        dwEvtMask := 0;
        //等待设置好的通信事件发生，由于有个 Os (Os : Toverlapped)，表示进行的
        //是 overlapped 等待，不会被这个等待堵塞住
        if not WaitCommEvent(hNewCommFile, dwEvtMask, @Os) then
        begin
            if ERROR_IO_PENDING = GetLastError then
                GetOverLappedResult(hNewCommFile, Os, dwTranser, True)
            end;
            if ((dwEvtMask and EV_RXCHAR) = EV_RXCHAR) then
            begin
                WaitForSingleObject(Post_event, INFINITE);
                // 等待允许传递 WM_COMMNOTIFY 通信消息
                ResetEvent(Post_Event);
                //处理 WM_COMMNOTIFY 消息，不再发送 WM_COMMNOTIFY 消息
                OK := PostMessage(Form1.Handle, WM_COMMNOTIFY, hNewCommFile, 0);
                // 传递 WM_COMMNOTIFY 通信消息
                if (not OK) then
                begin
                    MessageBox(0, 'PostMessage Error !', 'Notice', MB_OK);
                    exit;
                end;
            end;
        end;
    end;
end;

```

```

        end;
    end;
    CloseHandle(Os.hEvent);
    // 关闭重叠读事件对象
end;

//消息处理函数
procedure TForm1.WMCOMMNOTIFY(var Message :TMessage);
var
    CommState : ComStat;
    dwNumberOfBytesRead : Dword;
    ErrorFlag : Dword;
    InputBuffer : Array [0..1024] of Char;
begin
    // ClearCommError 回复通信错误信息并报告当前的通信设备状态。当通信错误
    //发生时调用此函数, 它会清除附加的 I/O 操作的设备错误标志
    if not ClearCommError(hNewCommFile,ErrorFlag,@CommState) then
        begin
            MessageBox(0,'ClearCommError !','Notice',MB_OK);
            PurgeComm(hNewCommFile,Purge_Rxabort or Purge_Rxclear);
            exit;
        end;
    if (CommState.cbInQue>0) then
        begin
            fillchar(InputBuffer,CommState.cbInQue,#0);
            // 接收通信数据
            if (not ReadFile( hNewCommFile,InputBuffer,CommState.cbInQue,
                                dwNumberOfBytesRead,@Read_Os )) then
                begin
                    ErrorFlag := GetLastError();
                    if (ErrorFlag <> 0) and (ErrorFlag <> ERROR_IO_PENDING) then
                        begin
                            MessageBox(0,'ReadFile Error!','Notice',MB_OK);
                            Receive :=False;
                            CloseHandle(Read_Os.hEvent);
                            CloseHandle(Post_Event);
                            CloseHandle(hNewCommFile);
                            exit;
                        end
                    end
                end
            end;
        end;
    end;
end;

```

```

        else
        begin
            WaitForSingleObject(hNewCommFile,INFINITE);
            // 等待操作完成, 等待设置好的 Event 的发生
            GetOverlappedResult(hNewCommFile,Read_Os,
                                dwNumberOfBytesRead,False);
        end;
    end;
    if dwNumberOfBytesRead>0 then
    begin
        Read_Os.Offset :=Read_Os.Offset+dwNumberOfBytesRead;
        ReceiveData := Read_Os.Offset;
        AddToMemo(InputBuffer,dwNumberOfBytesRead);
        // 处理接收的数据
    end;
end;
SetEvent(Post_Event);
// 允许发送下一个 WM_COMMNOTIFY 消息
end;

// 打开文件用于发送
procedure TForm1.Button1Click(Sender: TObject);
begin
    if OpenFileDialog1.Execute then
    begin
        Button3.Enabled :=False;
        Button4.Enabled :=False;
        RichEdit1.Lines.LoadFromFile(OpenDialog1.FileName);
        Form1.Caption := IntToStr(RichEdit1.GetTextLen);
    end;
    Button1.Enabled :=False;
end;
//发送数据
procedure TForm1.Button2Click(Sender: TObject);
var
    dcb : TDCB;
    Error :Boolean;
    dwNumberOfBytesWritten,dwNumberOfBytesToWrite,
    ErrorFlag,dwWhereToStartWriting : DWORD;

```

```

pDataToWrite : PChar;
Write_Os: TOverlapped;
begin
    Form1.Caption := "";
    hNewCommFile:=CreateFile('COM2',GENERIC_WRITE,0,nil,OPEN_EXISTING,
    FILE_FLAG_OVERLAPPED,0);
    // 打开通信端口 COM2
    if hNewCommFile = INVALID_HANDLE_VALUE then
        MessageBox(0,'Error opening com port!','Notice',MB_OK);
    SetupComm(hNewCommFile,1024,1024);
    //设置缓冲区大小及主要通信参数
    GetCommState( hNewCommFile,dcb);
    //设置 COM 口的 Data Control Block 的属性
    dcb.BaudRate :=9600;
    dcb.ByteSize :=8;
    dcb.Parity :=NOPARITY;
    dcb.StopBits := ONESTOPBIT;
    Error := SetCommState( hNewCommFile, dcb );
    if ( not Error) then
        MessageBox(0,'SetCommState Error!','Notice',MB_OK);
    dwWhereToStartWriting := 0;
    dwNumberOfBytesWritten := 0;
    dwNumberOfBytesToWrite :=RichEdit1.GetTextLen;
    if (dwNumberOfBytesToWrite=0) then
        begin
            ShowMessage("Text Buffer is Empty!");
            exit;
        end
    else
        begin
            pDataToWrite:=StrAlloc(dwNumberOfBytesToWrite+1);
            try
                RichEdit1.GetTextBuf(pDataToWrite,dwNumberOfBytesToWrite);
                Label1.Font.Color :=clRed;
                FillChar(Write_Os,SizeOf(Write_Os),0);
                // 为重叠写创建事件对象
                Write_Os.hEvent := CreateEvent(nil, True,False,nil);
                SetCommMask(hNewCommFile, EV_TXEMPTY);
                //用来表示对 EV_TXEMPTY 事件感兴趣, 有 Char 来到的时候系统会通知
            except
            end
        end
    end
end

```



```

Label1.Caption:='正在发送数据...!';
repeat
    Label1.Repaint;
    // 发送通信数据
    if not WriteFile( hNewCommFile, pDataToWrite[dwWhereToStartWriting],
        wNumberOfBytesToWrite,dwNumberOfBytesWritten, @Write_Os ) then
    begin
        ErrorFlag :=GetLastError;
        if ErrorFlag<>0 then
        begin
            if ErrorFlag=ERROR_IO_PENDING then
            begin
                WaitForSingleObject(Write_Os.hEvent,INFINITE);
                //等待设置好的 Event 的发生
                GetOverlappedResult(hNewCommFile,Write_Os,
                    dwNumberOfBytesWritten,False);
            end
            else
            begin
                MessageBox(0,'WriteFile 错误!','Notice',MB_OK);
                Receive :=False;
                CloseHandle(Read_Os.hEvent);
                CloseHandle(Post_Event);
                CloseHandle(hNewCommFile);
                exit;
            end;
        end;
    end;
    Dec( dwNumberOfBytesToWrite, dwNumberOfBytesWritten );
    Inc( dwWhereToStartWriting, dwNumberOfBytesWritten );
    // 写整个事情 ( Write the whole thing )
until (dwNumberOfBytesToWrite <= 0);
Form1.Caption:=IntToStr(dwWhereToStartWriting);
finally
    StrDispose(pDataToWrite);
end;
CloseHandle(hNewCommFile);
end;
Label1.Font.Color :=clBlack;

```

```

Label1.Caption:='发送成功!';
Button1.Enabled :=True;
Button3.Enabled :=True;
Button4.Enabled :=True;
end;

// 接收处理
procedure TForm1.Button3Click(Sender: TObject);
var
    Ok : Boolean;
    dcb : TDCB;
    com_thread: Thandle;
    ThreadID:DWORD;
begin
    ReceiveData :=0;
    Button1.Enabled :=False;
    Button2.Enabled :=False;
    RichEdit1.Clear;
    // 打开 COM2
    hNewCommFile:=CreateFile( 'COM2',GENERIC_READ,0,
                               nil, OPEN_EXISTING,FILE_FLAG_OVERLAPPED,0 );
    if hNewCommFile = INVALID_HANDLE_VALUE then
    begin
        MessageBox(0,'打开 COM 端口错误!','Notice',MB_OK);
        exit;
    end;
    Ok:=SetCommMask(hNewCommFile,EV_RXCHAR);
    if ( not Ok) then
    begin
        MessageBox(0,'SetCommMask 错误!','Notice',MB_OK);
        exit;
    end;
    SetupComm(hNewCommFile,1024,1024);
    // 设置缓冲区大小及主要通信参数
    GetCommState( hNewCommFile, dcb );
    dcb.BaudRate :=9600;
    dcb.ByteSize :=8;
    dcb.Parity :=NOPARITY;
    dcb.StopBits := ONESTOPBIT;

```

```

Ok := SetCommState( hNewCommFile, dcb );
if ( not Ok) then
    MessageBox(0,'SetCommState 错误!','Notice',MB_OK);
FillChar(Read_Os,SizeOf(Read_Os),0);
Read_Os.Offset := 0;
Read_Os.OffsetHigh := 0;
// 创建 Overlapped Read 事件
Read_Os.hEvent :=CreateEvent(nil,true,False,nil);
if Read_Os.hEvent=null then
    begin
        CloseHandle(hNewCommFile);
        MessageBox(0,'CreateEvent 错误!','Notice',MB_OK);
        exit;
    end;
//创建 PostMessage 事件
Post_Event:=CreateEvent(nil,True,True,nil);
if Post_Event=null then
    begin
        CloseHandle(hNewCommFile);
        CloseHandle(Read_Os.hEvent);
        MessageBox(0,'CreateEvent 错误!','Notice',MB_OK);
        exit;
    end;
Com_Thread:=CreateThread(nil,0,@CommWatch,nil,0,ThreadID);
// 建立通信监视线程
if (Com_Thread=0) then
    MessageBox(Handle,' CraeteThread 函数不起作用!',nil,mb_OK);
EscapeCommFunction(hNewCommFile,SETDTR);
Label1.Font.Color :=clRed;
Label1.Caption:=' 正在接收数据...! ';
end;

//停止通信处理
procedure TForm1.Button4Click(Sender: TObject);
begin
    Label1.Font.Color :=clBlack;
    Label1.Caption:='已停止通信';
    Form1.Caption := IntToStr(ReceiveData);
    Receive :=False;

```

```

    CloseHandle(Read_Os.hEvent);
    CloseHandle(Post_Event);
    CloseHandle(hNewCommFile);
    Button1.Enabled := True;
    Button2.Enabled := True;
end;

end.

```

6.2 基于 Windows TAPI 通信编程

微软视窗电话应用程序设计接口 (Telephony Application Programming Interface, 简称为 TAPI) 是 Microsoft 计算机电话集成 (CTI) 计划的核心, 为应用程序提供了一套支持电话通信的方法, 使程序员可以利用这个接口通过电话线使用多种计算机复杂的通信工作。下面将简单地描述一下 TAPI、应用程序是如何与 TAPI 打交道的, 最后介绍一个基于 TAPI 通信例子。

6.2.1 电话编程接口的简介

Microsoft 及 Intel 公司联合开发了 TAPI 这个应用程序接口供程序员控制和使用电话线。

TAPI 设备所具有的一些功能是: 直接与电话机相连、自动电话拨号、数据 (包括文件、传真和电子邮件) 传输、数据访问 (新闻、消息机构)、访问 Internet 或其他形式信息服务、组织会议呼叫、声音邮件、拨号鉴定、远程计算机控制以及通过电话线进行合作等。

TAPI 独立操作下一级的电话网络和设备。一个使用了 TAPI 的应用程序不用担心它连接在什么上, 而只需要自己与应用程序接口之间的交互。

应用程序通过 TAPI 与电话动态链接库进行交互。电话动态链接库是 Windows 操作系统的一部分并支持 TAPI, 它通过电话服务提供接口 (TSPI) 与服务提供者 (或服务提供商) 相联系。服务提供者可以被看作类似打印机驱动程序, 它们应该由电话硬件销售商编写并提供给用户, 以支持其在 Windows 下运行。

6.2.2 TAPI 主要函数和基于 TAPI 应用的基本步骤介绍

在编写 TAPI 程序时, 请仔细参考 WIN32 SDK 帮助的 TAPI 部分。Tapi.pas (Delphi 版的 TAPI 编程的头文件) 可从 Inprise 公司站点 (<http://www.inprise.com>) 获得, 也可从 <http://www.csdn.net/dev/Delphi/VCL/index.htm> 网页中的自由组件的源文件中获得。这里的 Tapi.pas 是 Alexander Staubo 写的。

所有的 Windows 98 通信围绕呼叫管理者而转, 它也称之为电话服务提供商 (Telephone Service Provider)。缺省的呼叫管理应用程序是电话拨号程序见图 6.3。Windows NT 中附带电

话应用程序 DIALER.EXE 的主界面如图 6.3 所示。DIALER.EXE 可以用来拨打电话、设置拨号属性、进行快捷拨号和建立连接。事实上，如果想使用任何高级的 TAPI 功能，必须自己创建一个定制的呼叫管理程序。



图 6.3 应用程序 DIALER.EXE

呼叫管理程序 (Call Manager) 可以处理许多工作。它必须创建一条线路，发出呼叫，处理异步回调和分配资源。

然而，如果不嫌麻烦，愿意创建一个定制的呼叫管理程序，则可以编写出一个功能强大的电话应用程序。下面详细介绍基于 TAPI 应用的基本步骤：初始化 TAPI、发出呼叫、处理回调和挂机并释放资源。

1. 初始化 TAPI

初始化 TAPI 有两个处理步骤：首先调用 `LineInitialize`，然后是 `LineNegotiateAPIVersion`。`LineInitialize` 为呼叫管理应用程序初始化 TAPI。它初始化 `tapi.dll` 的使用，注册应用程序的回调机制，并返回应用程序可用的逻辑线路设备数（使用 `lpdwNumDevs` 参数）。`Tapi.pas` 定义 `LineInitialize` 如下：

```
function lineInitialize(
    lphLineApp : LPHLINEAPP;
    hInstance : HINSTANCE;
    lpfnCallback : TLINECALLBACK;
    lpszAppName : LPCSTR;
    lpdwNumDevs : LPDWORD) : LONG;
```

调用 `lineInitialize` 函数作用为初始化线路，分配支持逻辑线路设备的使用而必需的一些内部资源。

`lphLineApp` 是指向 TAPI 使用实例的句柄。`hInstance` 是呼叫管理应用程序的实例 (instance)。`lpfnCallback` 是返回处理函数的地址。呼叫函数用来接收来自线路的异步响应。`lpszAppName` 是发出或者接收呼叫的应用程序名，如果设为 `null`，Windows 缺省使用呼叫应用程序的文件名。`lpdwNumDevs` 是可用线路设备数，`LineInitialize` 返回这个值（如果想查看

在自己的计算机上什么线路设备可用，打开控制面板，双击 Modems 图标即可，如图 6.4 所示）。

如果成功，LineInitialize 返回 0。如果 LineInitialize 返回 LINEERR_REINIT，应用程序应尝试再调用 LineInitialize 函数。LINEERR_REINIT 是一个非致命错误，它提示用户再试一次。其他错误值的解释，请参见 Win32 编程人员参考手册（Win32 Programmer's Reference）。为确定电话系统是否有配置问题，应检查 LineInitialize 的返回值。

LineNegotiateAPIVersion 告诉呼叫管理应用程序当前使用的是 TAPI 的哪种版本。因为 TAPI 有几个版本，每一个版本的功能都不同。TAPI 是要求版本协商（version negotiation）的几个 Windows API 的一员。Windows 95 使用 TAPI 1.4 版，Windows NT 使用 TAPI 2.0 版。最近 Microsoft 发布了一个 TAPI 3.0 版本。最新版中包括增加或者改变了的特性/函数。这些函数允许开发者使用一个特定的函数集，并保证 TAPI 的未来版本使用这些函数。



图 6.4 电话和调制解调器选项

```
function lineNegotiateAPIVersion(
    hLineApp : HLINEAPP;
    dwDeviceID : DWORD;
    dwAPILowVersion : DWORD;
    dwAPIHighVersion : DWORD;
    lpdwAPIVersion : LPDWORD;
    lpExtensionID : LPLINEEXTENSIONID) : LONG;
```

调用 lineNegotiateAPIVersion 函数实现版本协商，版本协商是为了应用程序在使用函数、结构及消息时，不必担心 TAPI 版本的新变化会产生不兼容或更糟的情况。

hLineApp 是指向线路 TAPI 使用实例的句柄，由 LineInitialize 函数返回的值。dwDeviceID 指定要用的设备。如果打算使用多个设备，则必须为每一个设备协商 TAPI 版本。LineInitialize 在参数 lpdwNumDevs 中返回可用的设备数。dwAPILowVersion 指定使用的 TAPI 最低版本，dwAPIHighVersion 指定使用的 TAPI 最高版本。Windows 将选择范围内的最高版本返回给

lpdwAPIVersion 参数, lpExtensionID 标识设备支持的 TAPI 扩展, 目前, UNIMODEM 不支持任何扩展, 可忽略 lpExtensionID 这个参数, 但不可省略。如果成功, 该函数返回为 0。

2. 发出呼叫

在初始化 TAPI 之后, 用函数 LineOpen 打开线路, 并用 LineMakeCall 拨打一个合适的号码。LineOpen 允许指定应用程序可以使用的通信函数, 包括回应呼叫、监视呼叫或者初始化呼叫。LineOpen 定义如下:

```
function lineOpen(
    hLineApp : HLINEAPP; dwDeviceID : DWORD;
    lphLine : LPHLINE; dwAPIVersion : DWORD;
    dwExtVersion : DWORD; dwCallbackInstance : DWORD;
    dwPrivileges : DWORD; dwMediaModes : DWORD;
    const lpCallParams : LPLINECALLPARAMS) : LONG;
```

调用 lineOpen 函数打开线路, 得到由 Windows 返回的线路设备句柄 hLine。

hLineApp 是指向 TAPI 使用实例的句柄。dwDeviceID 是个整数值。指定要打开的逻辑设备。lphLine 是指向被 Windows 打开的线路的句柄。dwAPIVersion 是函数 LineNegotiateAPIVersion 返回的 API 的版本值。dwExtVersion 是应用和服务提供商将使用的 extension 版本数 (正常情况下为 0)。dwCallbackInstance 不被 TAPI 使用, dwPrivileges 告诉管理员如何处理在线路上的呼叫, 表 6.6 列出了能够使用的值。lpCallParams 是指向 LINECALLPARAMS 结构的指针。表 6.6、表 6.7、表 6.8 的信息来源于在线的 Win32 程序员参考手册 (Win32 Programmer's Reference)。

表 6.6 TAPI 呼叫特权 dwPrivileges 影响应用程序的控制权

dwPrivileges 值	含义
LINECALLPRIVILEGE_NONE	应用程序呼出呼叫 (out-going calls)
LINECALLPRIVILEGE_MONITOR	应用程序监控呼入/呼出
LINECALLPRIVILEGE_OWNER	应用程序拥有 dwMediaModes 中指定类型的呼入呼叫
LINECALLPRIVILEGE_MONITOR	应用程序拥有 dwMediaModes 中指定类型的呼入呼叫
LINECALLPRIVILEGE_OWNER	如果不能作呼叫的主人, 则作监督者

表 6.7 TAPI 媒体模式 (Media Mode) 影响不同类型的呼叫

dwPrivileges 值	含义
LINEMEDIAMODE_UNKNOWN	应用程序处理未知类型的未分类的呼叫
LINEMEDIAMODE_INTERACTIVEVOICE	应用程序处理交互语音呼叫
LINEMEDIAMODE_AUTOMATEDVOICE	语音能量出现在呼叫中 (语音由自动应用程序处理)
LINEMEDIAMODE_DATAMODEM	应用程序处理数据 Modem 呼叫
LINEMEDIAMODE_G3FAX	应用程序处理组 3 传真呼叫

续表

dwPrivileges 值	含义
LINEMEDIAMODE_TDD	应用程序处理 TDD (Telephony Devices for the Deaf) 呼叫
LINEMEDIAMODE_G4FAX	应用程序处理组 4 传真呼叫
LINEMEDIAMODE_DIGITALDATA	应用程序处理数字数据呼叫
LINEMEDIAMODE_TELETEX	应用程序处理 teletex 呼叫
LINEMEDIAMODE_VIDEOTEX	应用程序处理 videotex 呼叫
LINEMEDIAMODE_TELEX	应用程序处理 telex 呼叫
LINEMEDIAMODE_MIXED	应用程序处理 ISDN 混合媒体呼叫
LINEMEDIAMODE_ADSI	应用程序处理 ADSI (Analog Display Services Interface) 呼叫
LINEMEDIAMODE_VOICEVIEW	呼叫媒体模式为 VoiceView

表 6.8

TAPI 使用的回调消息

消息	含义
LINE_ADDRESSSTATE	在打开线路中的地址状态已改变 (使用 LineGetAddressStatus 获取地址的当前状态)
LINE_CALLINFO	某种呼叫信息已改变 (使用 LineGetCallInfo 获取当前呼叫信息)
LINE_CALLSTATE	指定呼叫状态已改变 (使用 LineGetCallStatus 获取关于呼叫状态的信息信息)
LINE_CLOSE	指定线路设备已强制关闭, 相关的句柄不再合法
LINE_CREATE	新线路设备已创建
LINE_DEVSPECIFIC	提供与线路相关事件的设备特定信息, 一个地址或者呼叫
LINE_DEVSPECIFICFEATURE	提供与线路相关事件的设备特定信息, 一个地址或者呼叫
LINE_GATHERDIGITS	当前缓冲的数字搜集请求已终止或者已取消
LINE_GENERATE	当前的数字或者语音生成已终止或者已取消
LINE_LINEDEVSTATE	线路设备的状态已改变。使用 GetLineDevStatus 获取新状态
LINE_MONITORDIGITS	数字侦测到。用 LineMonitorDigits 控制
LINE_MONITORMEDIA	呼叫的媒体模式已改变。用 LineMonitorMedia 控制
LINE_MONITORTONE	语音侦测到。使 LineMonitorTones 用控制
LINE_REPLY	报告异步完成的函数调用的结果
LINE_REQUEST	报告来自另一个应用程序的新请求

dwMediaModes, 告诉 Windows 何种呼叫被回应或者被监控。注意, 这仅影响呼入的呼叫。因此, 可能会有 14 个应用程序处理呼叫, 一个对应一个类型。

lpCallParams 是指向 TLineCallParams 结构的指针。这仅在 LINEMAPPER 使用的情况下

使用, 否则此参数被忽略并被设为 nil。该参数设定线路的参数。如果成功, LineOpen 返回 0, 或者如果出错, 则返回一个负值的错误数。参见在线 Win32 编程人员资源, 其中列出可能的值及其引因。

目前为止已打开线路, 应用程序正等着拨打电话号码。为了处理这个呼叫, 可以使用 LineMakeCall 函数。呼叫可以管理。在 TAPI 中, 呼叫和线路被分别管理。LineMakeCall 函数定义如下:

```
function lineMakeCall(
    hLine : HLINE;
    lphCall : LPHCALL;
    lpszDestAddress : LPCSTR;
    dwCountryCode : DWORD;
    const lpCallParams : LPLINECALLPARAMS) : LONG;
```

调用 lineMakeCall 函数作用为生成电话呼叫, 在生成一次呼叫时, 至少要调用 lineMakeCall 函数一次。

hline 是指向用 LineOpen 打开线路的句柄, lphCall 是指向呼叫句柄的指针, lpszDestAddress 是欲拨的电话号码, 它的数据类型为 Pchar 且必须遵循电话号码数字规范, dwCountryCode 是呼叫者所在国家的号码, lpCallParams 是指向 LINECALLPARAMS 结构的指针。

最后, lpCallParams 是指向 TLineCallParams 结构的指针。其指定呼叫的特征。我们在 LineOpen 函数中看到这个结构, 该函数用来设定“线路”的参数。这里我们设定“呼叫”的参数。如果正进行数据呼叫 (data call), 必须创建这个结构并将 dwMediaMode 值设为 LINEMEDIAMODE_DATAMODEM。否则, 将该参数设为 nil。当此值设为 nil, LineMakeCall 自动发出交互语音呼叫 (interactive voice call) 而不管线路打开时设置何种特权 (privilege)。

LineMakeCall 是异步的。当相应的回调消息 LINE_REPLY 的 dwParam2 参数返回 0 时, 呼叫完成。如果有电话扩展 (telephone extension) 可以拨号, 那么在调用 LineMakeCall 后, 再调用函数 LineDial。不过, 在使用函数 LineDial 前, 必须等到 LineMakeCall 函数异步完成, LineDial 定义如下:

```
function lineDial(
    hCall : HCALL;
    lpszDestAddress : LPCSTR;
    dwCountryCode : DWORD) : LONG;
```

调用 lineDial 函数作用为生成电话呼叫, 用于分级拨号。前面讲到调用 lineMakeCall 函数, 调用后根据情况可多次调用 lineDial。这两个函数是异步的, 调用后立即返回, 直到以后才完成。呼叫建立后, 得到由 Windows 返回的呼叫句柄 hCall。

hCall 是指向由 LineMakeCall 返回的打开呼叫的句柄, lpszDestAddress 是欲拨的电话号码, 它的数据类型为 Pchar 且必须电话号码数字规范, dwCountryCode 是呼叫者的所在国家的号码。

3. 处理回调

回调函数处理来自于诸如 LineMakeCall 之类的异步操作的消息。记住,当写回调函数时,所有的回调发生在应用程序的上下文中。回调必须在动态链接库中或应用程序模块中。

4. 挂断

现在已经处理完数据传输,需要挂断电话。这是最容易的部分。如果仅有一个线路打开,LineShutDown 终止所有的呼叫并关闭 TAPI。

```
function LineShutDown(hLineApp: HLINEAPP): LONG;
```

调用 lineShutDown 释放为线路设备分配的资源。hLineApp 参数是线路 API 应用的使用句柄。

这个传入的参数是指向打开线路的句柄,其由调用 LineInitialize 返回。该函数终止 TAPI 实例,任何其后的呼叫必须重新初始化。然而,如果有几个呼叫打开,并被同一个应用程序管理,或者计划产生更多的呼叫,那么,需要分别地关闭每一个呼叫。可用两种方法:

(1) 在调用 LineDrop 之后调用 LineDeallocateCall,

(2) 调用 LineClose, LineClose 像调用 LineShutDown 一样,将释放所有打开的资源。这些函数直观明了,请参见在线 Win32 编程人员资源以获得更多的信息。

6.2.3 基于 TAPI 通信例子

现在至少能在 Windows API 函数中发送电话号码。然而,拨打一个电话号码往往涉及更多的 Windows API 函数。在打长途或外线时,可能需要加数字 1、8 或者 9。所以也需要一个可拨打的地址,可以由使用的电话系统组织。下面的 TAPI 通信例子将处理这些细节。

TAPI 通信例子中单元 TAPIUnit 的窗体如图 6.5 所示。Form1 窗体中包含了一个命名为 Memo 的 TMemo 组件,两个命名分别为 rbDefault 和 rbCallManager 的 TRadioButton 组件,两个命名分别为 btnDial 和 btnHangup 的 TButton 组件,一个命名为 ePhoneNum 的 TMaskEdit 组件。

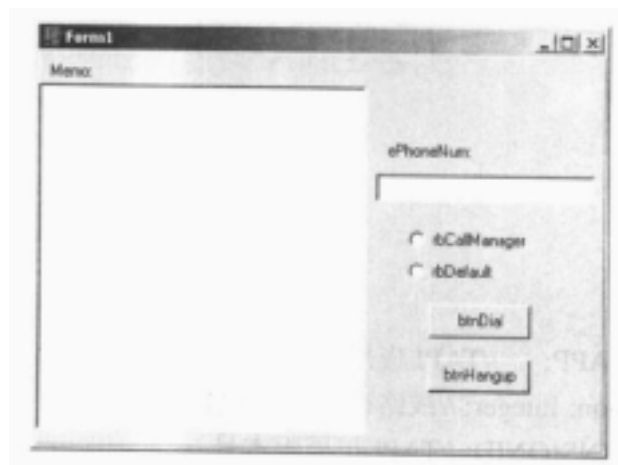


图 6.5 TAPIUnit 单元的窗体

单元 TAPIUnit 程序的代码如下:

```
unit TAPIUnit;
```

```

interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, Mask;
type

  TForm1 = class(TForm)
    Panel1: TPanel;
    ePhoneNum: TMaskEdit;
    Memo: TMemo;
    btnDial: TButton;
    btnHangup: TButton;
    rbDefault: TRadioButton;
    rbCallManager: TRadioButton;
    procedure btnDialClick(Sender: TObject);
    procedure FormCloseQuery(Sender: TObject;
      var CanClose: Boolean);
    procedure FormCreate(Sender: TObject);
    procedure btnHangupClick(Sender: TObject);
  private
    function TapiInitialize: Boolean;
    procedure CreateCallManager;
  public
    //在 public 处声明 ShutdownCallManager 函数
    function ShutdownCallManager: Boolean;
  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}
uses
  Tapi;
var
  FLineApp: HLINEAPP; //TAPI 应用句柄
  FNumDevs, FVersion: Integer; //线路设备数、TAPI 版本号
  FExt: TLINEEXTENSIONID; //TAPI 扩展版本号
  FLine: HLINE; //线路句柄
  FLineCallParams: TLineCallParams; //呼叫参数
  FLineOpen: Boolean;

```

```

FHCall: HCALL; //呼叫句柄
FCountryCode: Integer;
//初使化设备 ID 的值: 0..NumDevs
FDev: Integer;
const
  HiVer = $00020000; // 高版本为 2.0.
  LoVer = $00010004; // 低版本为 1.4

// 来自 line 的消息句柄
procedure LineCallBack(hDevice, dwMessage, dwInstance,
  dwParam1, dwParam2, dwParam3 : DWORD); stdcall;
begin
  with Form1, Memo.Lines do begin
    case dwMessage of
      // asynchronous 反映
      LINE_CALLSTATE:
        begin
          case dwParam1 of
            LINECALLSTATE_IDLE: //呼叫无效处理
              begin
                Add('LCB (LINE_CALLSTATE): ' + 'The call is idle; no call exists.');
                ShutdownCallManager;
              end;
            LINECALLSTATE_OFFERING:
              Add('LCB (LINE_CALLSTATE): ' + 'Call is being offered to the station.');
            LINECALLSTATE_ACCEPTED: Add('LCB (LINE_CALLSTATE): ' +
              'The call was offered and accepted.');
            LINECALLSTATE_DIALTONE: //检测到拨号音
              Add('LCB (LINE_CALLSTATE): ' + 'The call is receiving a dial tone.');
            LINECALLSTATE_DIALING: Add('LCB (LINE_CALLSTATE): Dialing ' +
              Form1.ePhoneNum.Text); //正在拨号
            LINECALLSTATE_RINGBACK: Add('LCB (LINE_CALLSTATE): ' +
              'The call is receiving ringback.');
            LINECALLSTATE_BUSY: //线路忙处理
              begin // 检测困难
                case dwParam2 of
                  LINEBUSYMODE_STATION:
                    Add('LCB(LINE_CALLSTATE): ' + 'Busy; called partys station busy.');
                  LINEBUSYMODE_TRUNK:

```

```

        Add('LCB (LINE_CALLSTATE): ' + 'Busy; trunk or circuit is busy. ');
LINEBUSYMODE_UNKNOWN:
        Add('LCB(LINE_CALLSTATE):'+ 'Busy;specific mode is unknown. ');
LINEBUSYMODE_UNAVAIL:
        Add('LCB(LINE_CALLSTATE):'+ 'Busy;specific mode unavailable. ');
    else
        Add('LCB (LINE_CALLSTATE): ' + 'Call receiving busy tone. ');
    end;
    ShutdownCallManager;
end;
LINECALLSTATE_SPECIALINFO:
    Add('LCB (LINE_CALLSTATE):'+ 'Special information sent by network. ');
LINECALLSTATE_CONNECTED:
    Add('LCB(LINE_CALLSTATE):'+ 'Call establish and connection made. ');
LINECALLSTATE_PROCEEDING: //呼叫正常处理
    Add('LCB (LINE_CALLSTATE): ' + 'Dialing completed; call proceeding. ');
LINECALLSTATE_ONHOLD:
    Add('LCB (LINE_CALLSTATE): ' + 'The call is on hold by the switch. ');
LINECALLSTATE_CONFERENCED:
    Add('LCB(LINE_CALLSTATE):Call is a multi-party conference call . ');
LINECALLSTATE_ONHOLDPENDCONF:
    Add('LCB(LINE_CALLSTATE):Call is on hold , add to conference. ');
LINECALLSTATE_DISCONNECTED:
begin
    Add('LCB (LINE_CALLSTATE): ' + 'The line has been disconnected. ');
    case dwParam2 of
        LINEDISCONNECTMODE_NORMAL:
            Add(#9 + 'A "normal" disconnect request. ');
        LINEDISCONNECTMODE_UNKNOWN:
            Add(#9 + 'Unknown reason for disconnect request. ');
        LINEDISCONNECTMODE_REJECT:
            Add(#9 + 'Remote user rejected the call. ');
        LINEDISCONNECTMODE_PICKUP:
            Add(#9 + 'Call picked up from elsewhere. ');
        LINEDISCONNECTMODE_FORWARDED:
            Add(#9 + 'Call was forwarded by switch. ');
        LINEDISCONNECTMODE_BUSY:
            Add(#9 + 'Remote users station is busy. ');
        LINEDISCONNECTMODE_NOANSWER:

```

```

        Add(#9 + 'Remote user station does not answer.');
```

LINEDISCONNECTMODE_BADADDRESS:

```

        Add(#9 + 'Destination address in invalid.');
```

LINEDISCONNECTMODE_UNREACHABLE:

```

        Add(#9 + 'Remote user could not be reached.');
```

LINEDISCONNECTMODE_CONGESTION:

```

        Add(#9 + 'The network is congested.');
```

LINEDISCONNECTMODE_INCOMPATIBLE:

```

        Add(#9 + 'Remote user's station ' + 'equipment is incompatible');
```

LINEDISCONNECTMODE_UNAVAIL:

```

        Add(#9 + 'Reason for the disconnect ' + 'is unavailable');
```

```

    end;
end;
LINECALLSTATE_UNKNOWN:
    Add('LCB (LINE_CALLSTATE): ' + 'The state of the call is not known.');
```

```

end;
end;
LINE_LINEDEVSTATE:
    case dwParam1 of
        LINEDEVSTATE_RINGING:
            Add('LCB (LINE_LINEDEVSTATE): ' + '(Ringing) Ring, ring, ring...');
```

LINEDEVSTATE_CONNECTED:

```

            Add('LCB (LINE_LINEDEVSTATE): Connected...');
```

LINEDEVSTATE_DISCONNECTED:

```

            Add('LCB (LINE_LINEDEVSTATE): Disconnected.');
```

LINEDEVSTATE_REINIT:

```

            // Line 设备已经被修改
            if (dwParam2 = 0) then
                begin
                    Add('LCB (LINE_LINEDEVSTATE): ' + 'Shutdown required');
```

 ShutdownCallManager;

```

                end;
            end;
end;
LINE_REPLY:
    if (dwParam2 = 0) then
        Add('LCB (LINE_REPLY): ' +
            'LineMakeCall completed successfully')
    else
        Add('LCB (LINE_REPLY): LineMakeCall failed');
```

```

    end;
  end;
end;
// -----
procedure TForm1.btnDialClick(Sender: TObject);
var
  ErrNo: longint;
  S: string;
begin
  if rbCallManager.Checked then
  begin
    btnDial.Enabled := False;    // dial 按钮使能属性设为 false
    btnHangup.Enabled := True;    // hangup 按钮使能属性设为 true
    CreateCallManager;
  end
  else
  begin //使用缺省的 call manager
    //没必要使这些控件无效, 缺省的 call manager 能处理事件 (the default call manager
    // handles everything)
    ErrNo := TAPIRequestMakeCall(PChar(ePhoneNum.Text), ',Some person, ');
    case ErrNo of
      0: S := 'success!'; // 成功
        TAPIERR_NOREQUESTRECIPIENT: S := 'TAPIERR_NOREQUESTRECIPIENT';
        TAPIERR_INVALIDDESTADDRESS: S := 'TAPIERR_INVALIDDESTADDRESS';
        TAPIERR_REQUESTQUEUEFULL: S := 'TAPIERR_REQUESTQUEUEFULL';
        TAPIERR_INVALIDPOINTER: S := 'TAPIERR_INVALIDPOINTER';
      else
        S := 'unknown value (' + IntToStr(ErrNo) + ')';
      end;
    Memo.Lines.Add('TapiRequestMakeCall returned: ' + S);
  end;
end;

procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
  //如果使用定制的 call manager, 释放资源
  if FLineOpen then
    CanClose := ShutdownCallManager;
end;

```

```

function TForm1.TapiInitialize: Boolean;
var
  ErrNo: Longint;
  S: string;
begin
  Result := False;
  //为了使用定制的 call manager, 初使化 TAPI
  FDev := 0;
  FCountryCode := 0;
  FVersion := 0;
  ErrNo := LineInitialize(@FLineApp, MainInstance,
                        LineCallback, '', @FNumDevs);
  //线路不能初始化处理
  case ErrNo of
    0:      Memo.Lines.Add('LineInitialize was successful');
    LINEERR_INVALIDAPPNAME:  S := 'LINEERR_INVALIDAPPNAME';
    LINEERR_OPERATIONFAILED: S := 'LINEERR_OPERATIONFAILED';
    LINEERR_INFILECORRUPT:   S := 'LINEERR_INFILECORRUPT';
    LINEERR_RESOURCEUNAVAIL: S := 'LINEERR_RESOURCEUNAVAIL';
    LINEERR_INVALIDPOINTER:  S := 'LINEERR_INVALIDPOINTER';
    LINEERR_REINIT:          S := 'LINEERR_REINIT - (try again)';
    LINEERR_NODRIVER:        S := 'LINEERR_NODRIVER';
    LINEERR_NODEVICE:        S := 'LINEERR_NODEVICE';
    LINEERR_NOMEM:           S := 'LINEERR_NOMEM';
    LINEERR_NOMULTIPLEINSTANCE: S := 'LINEERR_NOMULTIPLEINSTANCE';
  else
    S := 'Unknown Reason (' + IntToStr(ErrNo) + ')';
  end;
  //显示有多少设备可用
  Memo.Lines.Add('Devices available: '+IntToStr(FNumDevs));
  if (ErrNo <> 0) then begin
    Memo.Lines.Add('LineInitialize failed with error: ' + S);
    exit;
  end
else
  //协商 TAPI 版本号 TAPI1.4~TAPI2.0
  ErrNo := LineNegotiateAPIVersion(FLineApp, FDev, LoVer,
                                   HiVer, @FVersion, @FExt);

```



```

if (ErrNo <> 0) then
begin
    case ErrNo of
        LINEERR_BADDEVICEID:      S := 'LINEERR_BADDEVICEID';
        LINEERR_NODRIVER:         S := 'LINEERR_NODRIVER';
        LINEERR_INCOMPATIBLEAPIVERSION:
                                S := 'LINEERR_INCOMPATIBLEAPIVERSION';
        LINEERR_OPERATIONFAILED:   S := 'LINEERR_OPERATIONFAILED';
        LINEERR_INVALAPPHANDLE:    S := 'LINEERR_INVALAPPHANDLE';
        LINEERR_RESOURCEUNAVAIL:   S := 'LINEERR_RESOURCEUNAVAIL';
        LINEERR_INVALPOINTER:      S := 'LINEERR_INVALPOINTER';
        LINEERR_UNINITIALIZED:     S := 'LINEERR_UNINITIALIZED';
        LINEERR_NOMEM:             S := 'LINEERR_NOMEM';
        LINEERR_OPERATIONUNAVAIL:  S := 'LINEERR_OPERATIONUNAVAIL';
        LINEERR_NODEVICE:          S := 'LINEERR_NODEVICE';
    else
        S := 'Unknown Reason (' + IntToStr(ErrNo) + ')';
    end;
    // TAPI 版本不兼容
    LineShutDown(FLineApp);
    Memo.Lines.Add('LineNegotiateAPIVersion failed with error: ' + S);
end
else
    Result := True;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    FLineOpen := False;
    btnHangup.Enabled := False;    //使 hangup 控件无效
    if not TapiInitialize then
    begin
        Memo.Clear;
        Memo.Lines.Add('Failed to initialize TAPI');
        btnHangup.Enabled := False; //使 hangup 控件无效
        btnDial.Enabled := False;   //使 dial 控件无效
    end;
end;

```

```

procedure TForm1.btnHangupClick(Sender: TObject);
begin
    if rbCallManager.Checked then
        if not ShutdownCallManager then
            exit;
end;

procedure TForm1.CreateCallManager;
var
    S: string;
    ErrNo: longint;
begin
    Memo.Clear;
    //如果线路打开了，没必要初始化 TAPI
    if not FLineOpen then
        if not TAPIInitialize then
            begin
                Memo.Lines.Add('Failed to initialize TAPI');
                exit;
            end;
    //线路打开，得到由 Windows 返回的线路设备句柄 hLine
    ErrNo := LineOpen(FLineApp, FDev, @FLine, FVersion, 0, 0,
        LINECALLPRIVILEGE_NONE,
        LINEMEDIAMODE_INTERACTIVEVOICE, nil);
    case ErrNo of
        0: S := 'Line is open'; // 成功了
        LINEERR_ALLOCATED: S := 'LINEERR_ALLOCATED';
        LINEERR_BADDEVICEID: S := 'LINEERR_BADDEVICEID';
        LINEERR_INCOMPATIBLEAPIVERSION:
            S := 'LINEERR_INCOMPATIBLEAPIVERSION';
        LINEERR_INCOMPATIBLEEXTVERSION:
            S := 'LINEERR_INCOMPATIBLEEXTVERSION';
        LINEERR_INVALIDAPPHANDLE: S := 'LINEERR_INVALIDAPPHANDLE';
        LINEERR_INVALIDMEDIAMODE: S := 'LINEERR_INVALIDMEDIAMODE';
        LINEERR_INVALIDPOINTER: S := 'LINEERR_INVALIDPOINTER';
        LINEERR_INVALIDPRIVSELECT: S := 'LINEERR_INVALIDPRIVSELECT';
        LINEERR_NODEVICE: S := 'LINEERR_NODEVICE';
        LINEERR_LINEMAPPERFAILED: S := 'LINEERR_LINEMAPPERFAILED';
        LINEERR_NODRIVER: S := 'LINEERR_NODRIVER';
    end;

```

```

LINEERR_NOMEM:           S := 'LINEERR_NOMEM';
LINEERR_OPERATIONFAILED: S := 'LINEERR_OPERATIONFAILED';
LINEERR_RESOURCEUNAVAIL: S := 'LINEERR_RESOURCEUNAVAIL';
LINEERR_STRUCTURETOOSMALL: S := 'LINEERR_STRUCTURETOOSMALL';
LINEERR_UNINITIALIZED:   S := 'LINEERR_UNINITIALIZED';
LINEERR_REINIT:          S := 'LINEERR_REINIT';
LINEERR_OPERATIONUNAVAIL: S := 'LINEERR_OPERATIONUNAVAIL';
else
    S := 'LineOpen returned an unknown value of ' + IntToStr(ErrNo);
end;
Memo.Lines.Add('LineOpen reports: ' + S);
if (ErrNo <> 0) then
    exit
else
    FLineOpen := True;

    //创建和填入 LinCallParams 结构, 选择声音调用
    with FLineCallParams do
    begin
        dwTotalSize := sizeof(FLineCallParams);
        dwBearerMode := LINEBEARERMODE_VOICE;
        dwMediaMode := LINEMEDIAMODE_INTERACTIVEVOICE;
    end;
    //现在调用 LineMakeCall 函数
    ErrNo := LineMakeCall(FLine, @FHCall,
        PChar(ePhoneNum.Text), FCountryCode, @FLineCallParams);
    case ErrNo of
        0: S := 'LineMakeCall succeeded'; // 成功.
        LINEERR_ADDRESSBLOCKED: S := 'LINEERR_ADDRESSBLOCKED';
        LINEERR_BEARERMODEUNAVAIL: S := 'LINEERR_BEARERMODEUNAVAIL';
        LINEERR_CALLUNAVAIL: S := 'LINEERR_CALLUNAVAIL';
        LINEERR_DIALBILLING: S := 'LINEERR_DIALBILLING';
        LINEERR_DIALDIALTONE: S := 'LINEERR_DIALDIALTONE';
        LINEERR_DIALPROMPT: S := 'LINEERR_DIALPROMPT';
        LINEERR_DIALQUIET: S := 'LINEERR_DIALQUIET';
        LINEERR_INUSE: S := 'LINEERR_INUSE';
        LINEERR_INVALIDADDRESS: S := 'LINEERR_INVALIDADDRESS';
        LINEERR_INVALIDADDRESSID: S := 'LINEERR_INVALIDADDRESSID';
        LINEERR_INVALIDADDRESSMODE: S := 'LINEERR_INVALIDADDRESSMODE';
    end;

```

```

LINEERR_INVALBEARERMODE: S := 'LINEERR_INVALBEARERMODE';
LINEERR_INVALCALLPARAMS: S := 'LINEERR_INVALCALLPARAMS';
LINEERR_INVALCOUNTRYCODE: S := 'LINEERR_INVALCOUNTRYCODE';
LINEERR_INVALLINEHANDLE: S := 'LINEERR_INVALLINEHANDLE';
LINEERR_INVALLINESTATE: S := 'LINEERR_INVALLINESTATE';
LINEERR_INVALMEDIAMODE: S := 'LINEERR_INVALMEDIAMODE';
LINEERR_INVALPARAM: S := 'LINEERR_INVALPARAM';
LINEERR_INVALPOINTER: S := 'LINEERR_INVALPOINTER';
LINEERR_INVALRATE: S := 'LINEERR_INVALRATE';
LINEERR_NOMEM: S := 'LINEERR_NOMEM';
LINEERR_OPERATIONFAILED: S := 'LINEERR_OPERATIONFAILED';
LINEERR_OPERATIONUNAVAIL: S := 'LINEERR_OPERATIONUNAVAIL';
LINEERR_RATEUNAVAIL: S := 'LINEERR_RATEUNAVAIL';
LINEERR_RESOURCEUNAVAIL: S := 'LINEERR_RESOURCEUNAVAIL';
LINEERR_STRUCTURETOOSMALL: S := 'LINEERR_STRUCTURETOOSMALL';
LINEERR_UNINITIALIZED: S := 'LINEERR_UNINITIALIZED';
LINEERR_USERUSERINFOTOOBIG: S := 'LINEERR_USERUSERINFOTOOBIG';
else
  S := 'LineMakeCall returned an unknown value (' + IntToStr(ErrNo) + ')';
end;
Memo.Lines.Add('LineMakeCall reports: ' + S);
end;

function TForm1.ShutdownCallManager: Boolean;
var
  S: string;
begin
  Result := False;
  case LineShutdown(FLineApp) of
    0: begin
      S := 'success!';
      // LineShutDown 执行类似 LineClose 功能, 因此设置 FLineOpen 为 false
      FLineOpen := False;
      btnHangup.Enabled := False; // hangup 按钮使能属性设为 false
      btnDial.Enabled := True; // dial 按钮使能属性设为 true
      Result := True;
    end;
  LINEERR_INVALAPPHANDLE: S := 'LINEERR_INVALAPPHANDLE';
  LINEERR_NOMEM: S := 'LINEERR_NOMEM';

```

```
LINEERR_UNINITIALIZED:    S := 'LINEERR_UNINITIALIZED';  
LINEERR_RESOURCEUNAVAIL: S := 'LINEERR_RESOURCEUNAVAIL';  
else  
    S := 'Unknown value';  
end;  
Memo.Lines.Add('LineShutDown reports: ' + S);  
end;  
  
end.
```

本章小结

本章首先介绍了 Windows 98 和 Windows 3.x 的通信结构，接着介绍了串口通信的主要 API 函数，并给出了示例程序的源代码和分析，之后介绍了电话编程接口 (TAPI)，介绍了 TAPI 主要函数和通信程序的基本步骤，给出了基于 TAPI 应用通信例子及分析。

第 7 章 其他通信控件的使用

本章主要内容:

- ❑ 免费的 SPComm 控件
- ❑ TurboPower 的 APRO 通信控件

免费的 SPComm 控件具有很强的串口控制能力, 是个很有用的串口通信控件。

TurboPower 公司的 APRO 组件为 Delphi 和 C++ Builder 程序开发者提供了世界一流的通信工具套件, 笔者用的是 APRO 组件提供的 3.04 版本的工具套件。这个新版本完全支持 COM 端口控制、Winsock、文件传输和终端仿真, 增加了建立复杂商务电话应用程序 (比如语音信箱、Fax 广播器、呼叫应答和管理系统等) 的代码或控件。另外, APRO 3.04 完全支持与 AT 兼容的 ISDN Modem 和 RS-485。为了解 APRO 组件更多的信息, 读者可以访问 <http://www.turbopower.com/apro>。

本章介绍了 SPComm 控件和 TurboPower 公司的 APRO 和 APRO 2.x 的部分通信控件, 并给出了实例。

7.1 SPComm 控件的使用

利用免费的 SPComm 控件实现串口通信较简单。

下面介绍如何在 Delphi 中安装 SPComm 控件, 先打开 Delphi 5.0 集成开发环境, 选择菜单 “Component” 中的 Install Component 选项, 弹出如图 7.1 所示的窗口。

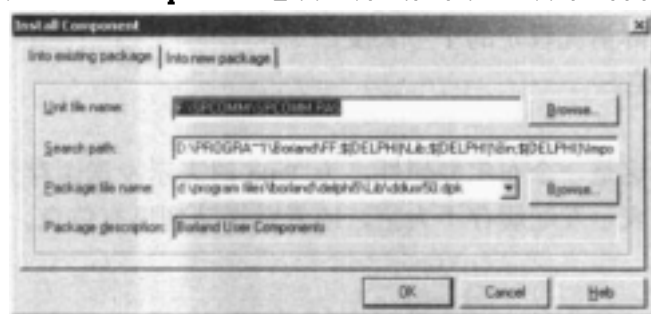


图 7.1 Component 中的 Install Component 窗口

在 Unit File name 处填写 SPComm 控件所在的路径, 其他各项可用默认值, 单击 OK 按钮, 进行控件安装。安装后, 在 System 控件面板中将出现一个红色控件 COM。现在就可以像 Delphi 自带控件一样使用 SPComm 控件了。SPComm 控件具有丰富的与串口通信密切相关的属性及事件, 提供了对串口的各种操作, 而且还支持多线程。

7.1.1 SPComm 的主要属性、方法和事件

下面介绍 SPComm 的主要属性、方法和事件。

1. 属性

SPComm 控件的属性如图 7.2 所示。



图 7.2 SPComm 控件的属性

CommName: 表示 COM1、COM2 等串口的名字。

BaudRate: 根据实际需要设定的波特率，在串口打开后也可更改此值，实际波特率随之更改。

ParityCheck: 表示是否需要奇偶校验。

ByteSize: 根据实际情况设定的字节长度。

Parity: 奇偶校验位。

StopBits: 停止位。

SendDataEmpty: 这是一个布尔型属性，为 **True** 时表示发送缓存为空，或者发送队列里没有信息；为 **False** 时表示发送缓存不为空，或者发送队列里有信息。

2. 方法

Startcomm 方法用于打开串口，当打开失败时通常会报错。错误主要有 7 种：

- (1) 串口已经打开
- (2) 打开串口错误
- (3) 文件句柄不是通信句柄
- (4) 不能够安装通信缓存
- (5) 不能产生事件

- (6) 能产生读进程
- (7) 不能产生写进程

StopComm 方法用于关闭串口，没有返回值。

WriteCommData 方法是个带有布尔型返回值的函数，用于将一个字符串发送到写进程，发送成功返回 True，发送失败返回 False。执行此函数将立即得到返回值，发送操作随后执行。该函数有两个参数，其中 pDataToWrite 参数是要发送的字符串，dwSizeofDataToWrite 参数是发送字符串的长度。

3. 事件

OnReceiveData: 当有数据输入缓存时将触发该事件，在这里可以对从串口收到的数据进行处理。其中 Buffer 参数中是收到的数据，BufferLength 参数是收到的数据长度。

OnReceiveError: 当接收数据出现错误时将触发该事件。

7.1.2 SPComm 控件的串口通信例子

下面介绍一个基于 SPComm 控件的串口通信的例子。

例子用于实现 PC 机与单片机 8051 之间的简单通信，这里只介绍 PC 机的通信程序。

例子中 PC 机与单片机 8051 之间的通信协议定为 PC 机到单片机 8051 一帧数据 6 个字节，8051 到 PC 一帧数据也为 6 个字节。当 PC 机发出 (F0, 01, FF, FF, 01, F0) 后，单片机 8051 能收到一帧 (F0, 01, FF, FF, 01, F0)，表示数据通信握手成功，两者之间就可以按照协议相互传输数据。

下面介绍本例子创建的步骤，首先创建一个新的工程 comm.dpr，把窗体的 Name 属性定为 FComm，把窗体的标题定义为串口通信测试，按照如图 7.3 所示添加控件（图中小矩形围住的控件即为 COMM1）。FComm 窗体中 Memo1 中用于显示发送和接收的数据。将新的窗体存储为 uComm.pas。

然后设定 FComm 窗体中 COMM1 属性，波特率 (BaudRate) 设为 4800，奇偶校验位 (ParityCheck) 设为 False，字节长度 (ByteSize) 设为 8，停止位 (StopBits) 设为 1，串口 (CommName) 设为 COM1。

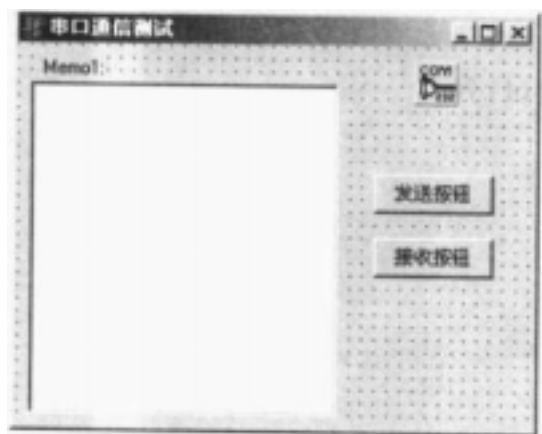


图 7.3 uComm 单元的窗体

下面给出 uComm 单元的源代码:

```
unit uComm

interface
uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    AbBase, AbZipOut, SPComm, StdCtrls;

type
    TFCComm = class(TForm)
        Comm1: TComm;
        Button1: TButton;
        Label1: TLabel;
        Button2: TButton;
        Memo1: TMemo;
        procedure FormShow(Sender: TObject);
        procedure FormClose(Sender: TObject; var Action: TCloseAction);
        procedure Button1Click(Sender: TObject);
        procedure Button2Click(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    FComm: TFCComm;
    viewstring:string;
    i:integer;
    rbuf,sbuf:array[1..16] of byte;
implementation
    {$R *.DFM}

    //自定义发送数据过程
    procedure senddata;
    var
        i:integer;
        commflg:boolean;
    begin
```

```
viewstring:="";
commflg:=true;
for i:=1 to 6 do
begin
    if not fcomm.Comm1.WriteCommData(@sbuf[i],1) then
    begin
        commflg:=false;
        break;
    end;
    //发送时字节间的延时
    sleep(2);
    viewstring:=viewstring+IntToHex(sbuf[i],2)+" ";
end;
viewstring:='发送'+ viewstring;
fcomm.Memo1.Lines.Add(viewstring);
fcomm.Memo1.Lines.Add(" ");
if not commflg then
    MessageDlg('发送失败 !',mterror,[mbytes],0);
end;

//打开串口
procedure TFComm.FormShow(Sender: TObject);
begin
    comm1.StartComm; //创建窗体时，将 comm1 控件打开
end;

//关闭串口
procedure TFComm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    comm1.StopComm; //关闭窗体时，将 comm1 控件关闭
end;

//发送按钮的单击事件
procedure TFComm.Button1Click(Sender: TObject);
begin
    sbuf[1]:=byte($f0); //帧头
    sbuf[2]:=byte($01); //命令号
    sbuf[3]:=byte($ff);
    sbuf[4]:=byte($ff);
```

```

    sbuf[5]:=byte($01);
    sbuf[6]:=byte($f0); //帧尾
    senddata;//调用发送函数
end;

//接收过程
procedure TFComm.Button2Click(Sender: TObject);
var
    StrReceive: string;
    Buffer: Pointer;
    BufferLength: Word;
begin
    SetLength(StrReceive, BufferLength);
    Move(Buffer^, PChar(StrReceive)^, BufferLength);
    // 函数 Move(const Source; var Dest; Count: Integer)
    // 函数 Move 作用为拷贝 byte
    Memo1.Lines.Add(StrReceive);
    //接收 RS232 的数据并显示 Memo1 上
    Memo1.Invalidate;
end;
end.

```

如果 Memo1 上显示发送 F0 01 FF FF 01 F0 和接收到 F0 01 FF FF 01 F0, 这表示串口已正确地发送出数据并正确地接收到数据, 则串口通信成功。

7.2 Turbopower 的 APRO 组件

组件板中的 APRO 组件如图 7.4 所示, 按图 7.4 的顺序分别有 TApdComPort 控件、TApdWinsockPort 控件、TApdRasDialer 控件、TApdRasStatus 控件、TApdFtpClient 控件、TApdFtpLog 控件、TApdDataPacket 控件、TApdScript 控件、TApdSModem 控件、TApdSL 控件、TApdStatusLight 控件、TApdProtocol 控件、TApdProtocolLog 控件、TApdProtocolStatus 控件、TApdTAPPager 控件、TApdSNPPPager 控件、TApdPagerLog 控件、TAdTerminal 控件、TAdTTYEmulator 控件、TAdVT100Emulator 控件。



图 7.4 组件板中的 APRO 组件

下面简单介绍几个 APRO 控件。

7.2.1 TApdComPort 控件

应用程序使用 TApdComPort 控件来控制串口。所有 I/O 串口访问均调用 TApdComPort 方法, 编写响应串行事件的事件处理器(Event Handler)。高层通信行为, 比如拨打一个 Modem 或传送一个文件, 可用 TApdComPort 与硬件交互。

对任何能影响它的通信调度程序生成 Delphi 事件的串口行为, Async Professional 均使用“Trigger”术语。共有 4 种类型的 Trigger。

data available: 接收的数据可用。

data match trigger: 一个特别的字符或字符串接收了。

status trigger: 一个状态事件已发生。

timer trigger: 一个计时器终止了。

TApdComPort 控件包含了多种管理 Trigger 的例程。Trigger 能被增加、激活, 修改且可设为无效。

调试一个通信程序有时要求不仅仅是一个传统的调试器。Async Professional 提供了跟踪和调度日志特性, 以帮助调试通信程序。

下面给出 TApdComPort 控件的重要属性介绍。TApdComPort 控件属性如图 7.5 所示。

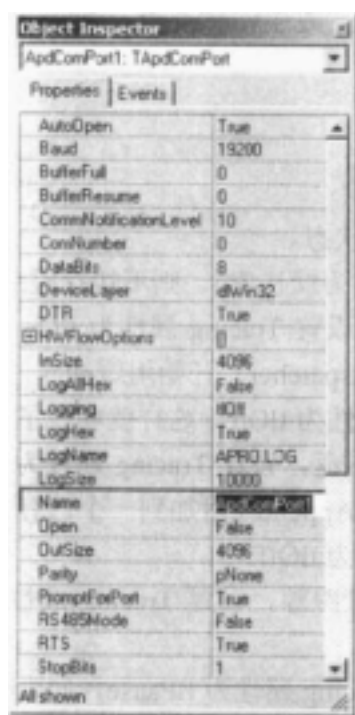


图 7.5 TApdComPort 控件属性

1. AutoOpen 属性

AutoOpen 决定了端口是否按要求自动打开。

如果 AutoOpen 是 True, 并且存取了一个要求具有打开的串口的方法或属性,

TApdComPort 控件将自动打开串口。如果 AutoOpen 是 False, 端口必须明确地打开 (通过设置 Open 属性为 True)。

2. TApiMode 属性

TApiMode 决定 TApdTapiDevice 控件是否能控制 TApdComPort 控件。

TApdTapiDevice 不能单独工作, 它必须联合 TApdComPort 一起工作。当创建一个 TApdTapiDevice 时, 它为 TApdComPort 查找窗体 (Form)。如果它找到一个, 它检查 comPort 控件的 TApiMode 属性以决定 TApdTapiDevice 是否能使用它。

如果 TApiMode 是 tmAuto (默认值), TApdComPort 控件对 TAPI 使用可用。TApdTapiDevice 保存一个到 TApdComPort 的指针并设置如下属性值:

```
ApdComPort.TApiMode := tmOn;
```

```
ApdComPort.AutoOpen := False;
```

```
ApdComPort.Open := False;
```

改变 TApiMode 设置为 tmOn 以表明关联的 TApdTapiDevice 正在控制 TApdComPort。设置 AutoOpen 和 Open 为 False, 因为它被打开或者被关闭 (TAPI 正在使用) 时, TApdComPort 不能再控制。

为了关闭 TAPI 模式, 或防止 TAPI 设备控制 TApdComPort, 设置 TApiMode 为 tmOff。为以后 TAPI 模式再使能, 设置 TApiMode 为 tmAuto 或 tmOn。读者须设置 AutoOpen 和 Open 为 False, 因为仅当 TApdTapiDevice 或 TApdComPort 被首先创建时, TApdTapiDevice 才自动设置这些属性。

tmNone 值没被使用。

3. Tracing 属性

Tracing 属性决定了当前跟踪状态。

当 Tracing 属性设为 tlOff (作为默认值), 则不执行跟踪。

为使能 (enable) 跟踪功能, 设置 Tracing 属性为 tlOn。这分配一个 2*TraceSize 字节大小的内部缓冲区, 通知调度器 (dispatcher) 开始用缓冲区。为使跟踪无效而不用写缓冲区的内容到磁盘文件, 设置 Tracing 属性为 tlOff。这样就释放了内部的缓冲区。

为了写跟踪缓冲区的内容到磁盘, 设置 Tracing 属性为 tlDump (重写命名为 TraceName 的文件, 或创建一个新文件) 或 tlAppend (附加到一个已有文件或创建一个新文件)。控件写内容到文件后, 设置 Tracing 属性为 tlOff。

为清除跟踪缓冲区内容并继续跟踪, 设置 Tracing 属性为 tlClear。控件清除跟踪缓冲区后, 设置 Tracing 属性为 tlOn。

为了临时暂停跟踪, 设置 Tracing 属性为 tlPause。如果要恢复跟踪, 设置 Tracing 属性为 tlOn。

请参考关于 Tracing 更多信息。

下面例子首先开启 Tracing, 然后转储 (Dump) 跟踪缓冲区到 APRO.TRC。

```
ApdComPort.Tracing := tlOn;
```

```
...
```

```
ApdComPort.TraceName := 'APRO.TRC';
```

```
ApdComPort.Tracing := tlDump;
```

异常: EbadArgument、EbadHandle、EinOutError、EoutOfMemory、EtracingNotEnabled。

4. Logging 属性

Logging 属性决定当前日志状态。

当 Logging 属性设为 tlOff (默认值), 没有执行日志功能。

为了启用日志, 设置 Logging 属性为 tlOn。这分配一个 LogSize 字节大小的内部缓冲区并通知调度器去开始用这个缓冲区。为了不将日志缓冲区的内容写到磁盘文件, 设置 Logging 属性为 tlOff。这样也释放了内部缓冲区。

为了将日志缓冲区的内容写到磁盘, 设置 Logging 属性为 tlDump 或 tlAppend。控件写文件之后, 设置 Logging 属性为 tlOff。

为清除缓冲区的内容并继续使用日志, 设置 Logging 属性为 tlClear。控件清空缓冲区之后, 设置 Logging 属性为 tlOn。

为临时暂停日志, 设置 Logging 属性为 tlPause。为了恢复日志功能, 设置 Logging 属性为 tlOn。

请参考关于调度日志功能的更多信息。

```
ApdComPort.Logging := tlOn;
```

...

```
ApdComPort.LogName := 'APRO.LOG';
```

```
ApdComPort.Logging := tlDump;
```

异常: EbadHandle、EinOutError、EloggingNotEnabled、EoutOfMemory。

5. DTR 属性

DTR 决定当前“Data Terminal Ready”信号 (DTR) 的状态。

一些类型的远程设备要求在传送前升高这个信号。例如, 除非 PC (计算机) 升高 DTR 信号, 否则默认配置的 Modem 不会传送数据。

下面例子表明在打开端口之后, 降低 DTR 信号, 其后升高 DTR 信号。

```
ApdComPort := TApdComPort.Create(Self);
```

```
ApdComPort.Open := True;
```

```
ApdComPort.DTR := False;
```

...

```
ApdComPort.DTR := True;
```

异常: EbadArgument。

6. ComNumber 属性

ComNumber 决定了 TApdComPort 控件使用的串口号 (Com1、Com2、...)。

ComNumber 并不验证串口号的有效性。当打开端口时, Windows 通信驱动程序将决定串口号是否有效, 如果无效, 则出错。

当改变 ComNumber 的属性时, 如果端口打开, 关闭已有的端口, 用新串口号重新打开。在这个操作中, 维持 Trigger (触发器)。

当使用 TAPI 和 Winsock 设备层 (Device Layer) 时, 这个属性被忽略。

下面例子在运行期间创建、配置、打开一个 ComPort 控件。

```
ApdComPort := TApdComPort.Create(Self);
```

```
ApdComPort.ComNumber := 1;
```

```
ApdComPort.Baud := 9600;
```

```
ApdComPort.Parity := pNone;
```

```
ApdComPort.DataBits := 8;
```

```
ApdComPort.StopBits := 1;
```

```
ApdComPort.Open := True;
```

7. Open 属性

Open 决定是否打开端口, 是否用当前端口的所有属性初使化该端口。

在 ComPort 控件能发送或接收字符前, Open 必须设定为 True。如果 AutoOpen 属性设为 True, 在许多情形下: 调用任何 I/O 方法或者属性, 或者当使用 TApdComPort 的控件装载时, ComPort 控件将自动打开自己。

当 Open 属性设为 True 时, TApdComPort 控件将用所有当前属性设置来分配输入和输出缓冲区, 打开物理端口, 初使化线路设置 (Line Setting) 和流量控制 (Flow Control) 设置, 并使 Tracing 和 logging 有效或无效 (enable or disable tracing and logging)。然后为低级端口注册一个 trigger, 其首先查看所有 trigger 事件, 并将控制传递给适当的 OnTriggerXxx 事件处理器。

当 Open 属性设为 False 时, TApdComPort 将关闭 Tracing 和 Logging (通过设置相关的属性为 tlDump, 如果已经缓冲了信息, 它将创建一个输出文件), 关闭端口, 重新分配输入和输出缓冲区。

当 Open 属性已经为 True 时, 设置 Open 属性为 True 是无害的; 或者当 Open 属性已经为 False 时, 设置 Open 属性为 False 是无害的。

8. StopBits 属性

StopBits 属性决定端口的停止位 (Stop Bit) 的个数。

可接受的值为 1 和 2。如果 DataBits 等于 5, 一个请求 2 个停止位被解释为一个请求 1.5 个停止位 (标准数据大小)。

当改变 StopBit 属性时, 如果端口已打开, 线路参数立即更新。在把 StopBit 属性传递给通信驱动程序之前, StopBit 不校验所赋的值, 驱动程序可能拒绝这个值, 并将导致一个异常。

异常: EBadArgument、EBadHandle。

9. DataBits 属性

DataBits 决定端口的数据位的个数。

可接受的值是 5、6、7、8。

当改变 DataBits 属性时, 如果端口已打开, 线路参数立即更新。把 DataBits 传递给通信驱动程序之前, DataBits 不校验所赋的值, 驱动程序可能拒绝这个值, 并将导致一个异常。

异常: ENotSupported。

10. Parity 属性

奇偶位决定端口的奇偶校验模式。

当 Parity 被改变时，如果端口已打开，线路参数立即更新。把 Parity 传递给通信驱动程序之前，Parity 不校验所赋的值，驱动程序可能拒绝这个值，并将导致一个异常。

异常：EBadHandle、ENotSupported。

11. Baud 属性

Baud 决定端口使用的波特率。

通常可接受的波特率值包括 300、1200、2400、4800、9600、19200、38400、57600 和 115200。

当波特率（Baud）被改变时，如果端口已打开，线路参数立即更新。把 Baud 传递给通信驱动程序之前，Baud 不校验所赋的值，驱动程序可能拒绝这个值，并将导致一个异常。

用对象观察器（Object Inspector）输入一个波特率或调用 SelectBaudRate 属性编辑器，其提供了一个标准波特率的下拉列框（Drop-Down List Box）。

异常：ENotSupported。

12. OnConnectionStatus 事件

当 Modem 状态改变时，OnConnectionStatus 定义了调用的事件处理器。

这个事件处理所有 TApdSModem 状态改变。下面 Modem 状态将触发 OnConnectionStatus 事件：

smsReady	空闲并就绪
smsInitialize	开始初始化进程
smsInitializeTimeout	等待初始化响应超时
smsAutoAnswerBackground	autoanswer 模式，没振铃接收
smsAutoAnswerWait	autoanswer 模式，等待第 N 个铃声
smsAnswerWait	回应呼叫，等待连接
smsDialWait	拨号呼叫，等待连接
smsDialCycle	重试拨号尝试的时间
smsNoDialTone	当拨号尝试时，Modem 报告没有拨号音码
smsConnected	处理连接过程
smsHangup	开始挂断过程
smsCancel	开始取消过程

在 OnConnectionStatus 事件处理器，用 TApdSModemStatusInfo 类的方法来获取 Modem 状态和那个情况的合适的状态信息。

7.2.2 TApdRasDialer 控件

TApdRasDialer 控件提供了 Microsoft 远程访问服务（Remote Access Service RAS）API 的接口。通过 Window 拨号网络（Dial-Up Networking），这个控件主要用来与一个远程计算机建立和终止联接，然而它也用来处理 RAS 电话簿项（RAS Phonebook Entries）和枚举活动

连接 (Active Connection)。

TApdRasDialer 要求 RAS 已经安装在应用程序将运行的计算机上。如果没有安装 RAS, 当 TApdRasDialer 函数调用时, 将弹出异常 `ecRasLoadFail`。

通过 `Dial` 和 `DialDlg` (Windows NT) 方法执行拨号。用 `Dial` 方法, 同步和异步拨号选项都有效。Hangup 方法终止呼叫。CreatePhonebookEntry、DeletePhonebookEntry、EditPhonebookEntry、ListEntries 和 PhonebookDlg (Windows NT) 方法操作 Phonebook 项 (Entry)。

TApdRasDialer 控件的属性如图 7.6 所示。

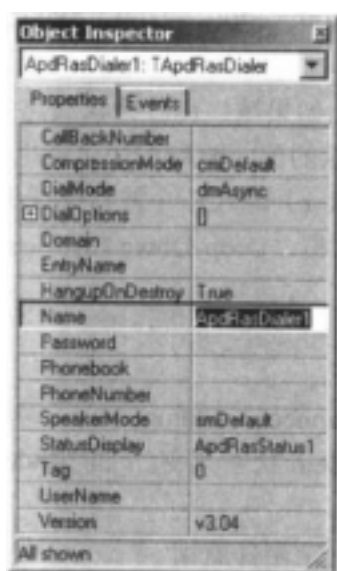


图 7.6 TApdRasDialer 控件属性

`GetDialParameters` 和 `SetDialParameters` 方法可以访问一个特殊呼叫拨号参数。对于 Windows NT 操作系统的计算机, `MonitorDlg` 能显示 RAS 连接的状态。

1. Password 属性

指定一个包含用户密码的字符串。

密码为用户访问远程计算机的授权密码。

2. domain 属性

指定一个字符串, 其中包含授权 (Authentication) 发生的域。

空字符串指定域 (远程访问服务器在域中是一个成员)。一个星号 (*) 指定存储在 Phonebook 中的域。

3. Dial 属性

在 RAS 客户端和 RAS 服务器端之间, 用 `Dial` 建立一个远程访问服务 (RAS) 连接。如果一个连接错误发生, 这个连接将自动挂断。

在异步方式拨号 (`ialMode = dmAsync`) 中, 在连接建立前, 拨号立即返回。连接进度通过 `OnDialStatus`、`OnDialError` 和 `OnConnected` 事件表达。另外, 如果 `StatusDisplay` 指定

TApdRasStatus 控件, Status 控件显示一个拨号状态对话框, 直到建立或取消连接。

在同步拨号期间 (DialMode 设为 dmSync), 拨号不会返回, 直到连接尝试已经成功完成或失败。没事件触发, 因此拨号函数结果必须检查以决定连接状态。

一个不同于 ecOK 的返回值表明一个错误发生并且返回的值是个错误代码。这个值传递给 GetErrorText 可以获得错误的描述信息。

通过 ConnectState 属性, 可获得连接状态信息, 直到应用程序调用 HangUp 终止连接。在一个连接已成功建立后, 应用程序最后必须调用 HangUp。

拨号不显示一个登录 (Logon) 对话框。这当前通过远程网络应用程序完成。应用程序负责设置拨号属性。

7.2.3 TApdRasStatus 控件

TApdRasStatus 控件提供一个标准 RAS 拨号状态对话框, 其中含有一个可在任何时间中断拨号的 Cancel 按钮。为了使用它, 只要创建一个实例, 将之赋值给 TApdRasDialer 控件的 StatusDisplay 属性,

TApdRasStatus 没有必须调用的方法或必须调整的属性。更改外观和窗口的布置, 可以修改 Ctl3D 设置和 Position 属性。

7.2.4 TApdSModem 控件

TApdSModem (简单 Modem) 控件结合 TApdModemDBase、TApdModem、TApdModemDialer 和 TApdDialerDialog 控件的特征而成为具有更简单界面的单一的控件。

TApdSModem 控件属性如图 7.7 所示。



图 7.7 TApdSModem 控件属性

TApdSModem 集成了来自 Modem 数据库的 Modem 选项和显示当前 Modem 状态的对话框。这样控件包含了许多特征, 而控件没有变得复杂。当大部分的属性仍在后台, 用户可能并不需要改变或访问它们。

除属性区列出的属性, TApdSModem 还有其他高级属性, 可能对高级用户很有用。这些属性只在运行期可用。

1. SelectModem 属性

SelectModem 属性显示了一个允许用户选择 Modem 的对话框。

用户能选择一个 Modem 和指定连接的串口。选择的 Modem 的配置设置被读进 TApdSModem 的内部变量。

2. Dial 属性

Dial 属性拨打指定的电话号码。

Dial 属性初始化 Modem 控件的 Trigger 以等待连接, 然后拨打 PhoneNumber 属性指定的号码。当连接建立时, OnConnectionStatus 被调用 (处于 smsConnected 状态)。

首次拨号尝试后, 如果连接没建立, AutoRetry 指定是否自动重试, 在任何时间调用 Cancel, 拨号操作均被取消。如果 ShowStatus 为 True, 在拨号过程中显示状态对话框。

下面的例子通过 Modem 拨号。

```
ApdSModem1.PhoneNumber := '555-1212';
```

```
ApdSModem1.Dial(False);
```

异常: EModemBusy、EModemNotResponding、EModemRejectedCommand。

3. PhoneNumber 属性

PhoneNumber 属性决定了拨打的电话号码。

这个属性设置的电话号码, 其拨号呼叫时使用。在拨号过程中, PhoneNumber 属性可以改变, 允许用户循环拨打电话号码列表中的电话, 直到连接建立。例如: 如果拨打 555-1212 没有连接, 可用在 OnConnectionStatus 里的 smsDialCycle 状态来改变电话号码 (而非重新拨打 555-1212)。

4. ModemIniName 属性

ModemIniName 是个包含 Modem 配置信息的文件名。

Modem 数据库文件包含一系列 Modem 和它们的配置字符串。默认的 Modem Database 文件是 AWMODEM.INI, 由 Async Professional 提供。当 SelectModem 调用显示用户能选择的一系列 Modem, 使用 ModemIniName 这个文件; 在设计时, 当修改 ModemName 属性值时, 也使用 ModemIniName 这个文件来显示 Modem 列表。

5. AutoAnswer 属性

在指定数量的振铃之后, AutoAnswer 准备 Modem 去应答呼叫。

AutoAnswer 设置合适的变量和 trigger (触发器) 后, 控制返回到程序, Modem 控件在后台监视呼入 (Incoming Call)。如果接收到 RingCount 振铃结果, 呼叫回应。如果 ShowStatus 为 True, 在回应期间, 显示一个状态对话框。

调用 Cancel, 取消自动响应模式。不管 TApdSModem 是否在后台等待呼入或者 Modem 当前正在回应呼叫, 自动响应模式均被取消。

调用 AutoAnswer 并不打开外部 Modem 的自动应答 (Auto Answer AA) 灯。AutoAnswer 方法没有用 Modem 的自动应答特性。

下面例子为在两次振铃后告诉 Modem 摘机 (Pick Up)。

```
ApdSModem1.AutoAnswer(2);
```

7.2.5 TApdModem 控件

TApdModem 控件为访问 Modem 提供了与设备无关的例行程序。它提供初使化、配置、拨号、应答和其他公共的 Modem 功能。

TApdModems 有直接对应于 TModemInfo 结构中域的属性。这些属性决定了哪些字符串送到 Modem, 哪些字符串期待返回。

TApdModem 控件有个 ModemInfo 属性, ModemInfo (TModemInfo 类型) 可用来一次设定所有的命令和响应字符串。如果从 Modem 数据库或 INI 文件中读 Modem 配置数据, 这个属性特别有用。

另外, TApdModem 有两个特别属性: ErrorCorrectionTags 和 DataCompressionTags, 用来设置 Modem 返回的字符串(其显示错误更正和/或数据压缩特征)。这些属性是 TtagSet 类型。

1. DataCompressionTags 属性

DataCompressionTags 属性是一系列字符串, 在拨号或者回应过程中, Modem 能返回这些字符串, 以尝试指出在连接时可用的数据压缩特征。

DataCompressionTags 属性是个 TTagSet 类型。TTagSet 是个简单的类, 在对象观察器中, 其惟一的目的是使能数据压缩标签的编辑, 5 个可能的数据压缩标签(String1、String2、String3、String4 和 String5) 作为 DataCompressionTags 属性的子属性出现。标签字符串的最大长度为 21 个字符。

下列例子设置 Modem 在连接时查找 V.42 数据压缩。

```
ApdModem1.DataCompressionTags.String1 := 'V42BIS';
```

异常: EModemBusy、EoutOfMemory。

2. ConfigCmd 属性

ConfigCmd 属性决定了配置 Modem 的字符串。

ConfigCmd 属性决定了 Modem 的一般设置。当 Configure 方法调用时, 发送 ConfigCmd 属性。

与 TApdModem 控件发送给 Modem 的其他的命令字符串不同的是 ConfigCmd 能包含多个命令(捆绑在一起)。TApdModem 分别校验和传送每个命令(可达到最大长度为 255 字符)。串中的每个命令被 CmdSepChar 分离, 默认的字符为 “|”。

异常: EmodemBusy、EoutOfMemory。

3. AnswerCmd 属性

AnswerCmd 属性决定送到 Modem 去准备响应一个呼入的字符串。

当调用 Answer 和 AutoAnswer 方法, 字符串送到 Modem。这个字符串通常用 “ATA^M”。
Exceptions: EmodemBusy、EoutOfMemory。

4. LastString 属性

LastString 属性是 Modem 接收到的最后字符串。

通常, TApdModem 控件内部处理所有响应, 并通过事件报告结果。然而, 通过检查 LastString 属性, 也可能人工地检查 TApdModem 处理的文本。

例如, 如果 Modem 控件将“ATZ”命令发送到 Modem 中, 则 Modem 应返回“OK”字符。这时, Modem 控件生成 OnModemOK 事件, 并且 LastString 为“OK”。

如果 Modem 控件在拨号, 当一个连接成功建立, Modem 返回一个类似“CONNECT 19200”的字符串。Modem 控件生成一个 OnModemConnect 事件, LastString 是“CONNECT 19200”。

下面例子示范了 LastString 的使用。

```
procedure TForm1.ApdmModem1.CommandProcessed(Sender : TObject;
                                             WhatHappened : TModemStatus;
                                             Data : LongInt);
begin
    StatusLabel.Caption := 'Modem Status: ' + ApdmModem1.LastString;
end;
```

5. HangupCmd 属性

HangupCmd 属性决定了送到 Modem 来挂断线路的字符串。

当调用 Hangup 方法时, 该字符串送到 Modem。在 AWMODEM.INI 数据库, 它有个默认的值, 即“+++~~~ATH0^M”。

Exceptions: EmodemBusy、EoutOfMemory。

6. OnConnectFailed 事件

当一个连接尝试失败, OnConnectFailed 定义了一个调用的事件处理器。

不论因为何种原因 (例如: 忙音、握手错误等) 而造成一个连接尝试失败, 最终, OnConnectFailed 均被调用。

调用 OnConnectFailed 后, 再使用 Modem 控件是安全的。如果在事件处理器内部调用一个 TApdModem 方法, 那么不返回 EModemBusy 异常。

下面例子示范了 OnConnectFailed 和 OnModemIsConnected 的使用。

```
procedure TMyForm.ModemIsConnected(Sender : TObject);
begin
    ApdComPort1.Output := 'I connected to you.';
    ApdModem1.Hangup;
end;

procedure TMyForm.ConnectFailed(Sender : TObject);
begin
    //一个连接失败后, 重新初使化 Modem
    ApdModem1.Initialize;
end;
```

7.2.6 TApdSLController 控件

TApdSLController 控件监视 TApdComPort 控件的状态，并改变一个或多个 TApdStatusLight 控件的状态来反映那个状态。

TApdSLController 有能力监视端口线路信号（DCD、DTR、CTS 和 RI），线路断开和错误以及是否当前接收或传送数据。

为了使用 TApdSLController 控件，并让控制器去监视线路，首先对于每个线路条件均创建一个 TApdStatusLight 控件。下一步，放一个 TApdSLController 控件在窗体（Form）中，将之链接到要监视的 TApdComPort 控件。下一步，链接这个控制器的 Light 属性到 Status Light 控件。最后，设置控制器的 Monitoring 属性为 True。TApdSLController 控件属性如图 7.8 所示。

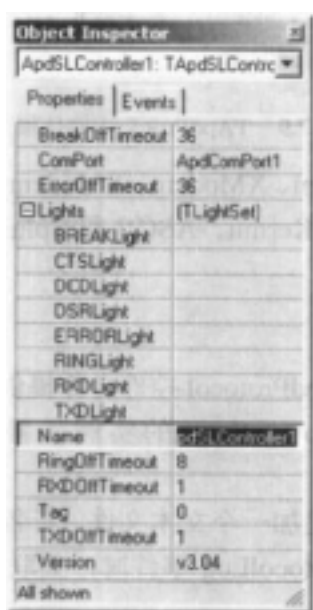


图 7.8 TApdSLController 控件属性

7.2.7 TApdStatusLight 控件

TApdStatusLight 是一个显示两个位图或两种不同颜色的简单控件。这个控件的 Lit 属性决定两个状态哪个被显示，它的 Glyph 属性决定使用位图还是纯粹用颜色（Solid Color）。

这个控件与 TApdSLController 控件手拉手地工作。TApdSLController 对串口状态改变和各个 TApdStatusLight 控件的 Lit 属性改变作出反应，以反映串口的状态。

7.2.8 TApdProtocol 控件

在一个综合控件里 TApdProtocol 控件实现所有的 Async Professional 文件传输能力。TApdProtocol 控件属性如图 7.9 所示。



图 7.9 TApdProtocol 控件属性

TApdProtocol 控件能用 ZModem、XModem、XModemCRC、XModem1K、XModem1KG、YModem、YModem、YModemG、Kermit、ASCII 和 Bplus 协议传输文件。

7.2.9 TApdProtocolLog 控件

TApdProtocolLog 控件是与 TApdProtocol 控件合作提供自动协议日志服务的一个小类型。所要做的只是创建一个 TApdProtocolLog 的实例，将之赋值给 TApdProtocol 控件的 ProtocolLog 属性。

TApdProtocolLog 控件创建或追加一个文本文件，其名字由 HistoryName 的属性给出。每次生成 TApdProtocol 控件的 OnProtocolLog 事件时，关联的 TApdProtocolLog 实例打开这个文件，写一行，并关闭这个文件。

7.2.10 TApdProtocolStatus 控件

TApdProtocolStatus 控件属性如图 7.10 所示。



图 7.10 TApdProtocolStatus 控件属性

TApdProtocolStatus 控件是实现标准协议状态显示的 TApdAbstractStatus 的继承者。要做的是创建一个实例，将之赋值给 TApdProtocol 控件的 StatusDisplay 属性。

TApdProtocolStatus 控件可以继承 TApdAbstractStatus 控件的所有抽象方法。TApdProtocolStatus 没有必须调用的方法或必须调整的属性。可去改变 Ctl3D 的设置和 Position 属性以更改外观和窗口的布置。

7.3 Turbopower 的 APRO 2.x 组件

下面简单介绍 APRO 2.x 组件。组件板中的 APRO 2.x 组件如图 7.11 所示，按图 7.11 的顺序分别有 TApdTerminal 控件、TApdBPTerminal 控件、TApdEmulator 控件、TApdKeyboardEmulator 控件、TApdIniDBase 控件、TApdModemDBase 控件、TApdModem 控件、TApdPhonebook 控件、TApdPhonebookEditor 控件、TApdPhoneNumberSelector 控件、TApdModemDialer 控件、TApdDialerDialog 控件。



图 7.11 APRO 2.x 组件

下面简单介绍几个 APRO 2.x 控件。

7.3.1 TApdModemDBase 控件

TApdModemDBase 控件提供读写 Async Professional 的 Modem 数据库文件格式的服务。Async Professional 包括一个 AWMODEM.INI 的文件，它包括 Modem 配置数据，适合于多种流行 Modem。TApdModemDBase 控件属性如图 7.12 所示。



图 7.12 TApdModemDBase 控件属性

Modem 数据库, 如同 TApdIniDBase.INI 文件数据一样, 有能力增加记录、改变记录、删除记录和装载文件中的所有记录进入一个 TString 类。

Modem 数据库的记录结构是一个 TModemInfo 记录。

1. FileName 属性

FileName 决定了 Modem 数据库的文件名。

如果 FileName 没指定一个目录, Windows 的 INI 文件服务在主 Windows 目录里查找或创建文件。如果 FileName 指定的目录不存在, 将引发一个 EiniWrite 异常。

异常: EdataTooLarge、EiniWrite、Einternal、EkeyTooLong、EoutOfMemory。

2. Open 属性

Open 决定一个 Modem 数据库是否打开进行读写。

当 Open 设为 True, 控件打开一个存在的 Modem 数据库文件。如果数据库文件不存在, 数据库文件将被创建, 并被写入一个初始部分 (基于一个 Modem 数据库的标准格式)。

当 Open 设为 False, 关闭数据库文件, 释放内部数据结构。

如果调用需要访问数据库文件的方法, Modem 数据库将自动打开。

下面例子打开一个 Modem 数据库, 然后关闭。

```
DB := TApdModemDBase.Create(Self);
DB.FileName := 'C:\ASYNCPRO\AWMODEM.INI';
DB.Open := True;
```

...

```
DB.Open := False;
```

异常: EdataTooLarge、EiniWrite、Einternal、EkeyTooLong、EoutOfMemory。

3. ReadFromIni 方法

ReadFromIni 从一个单独的 INI 文件读 Modem 数据。

在 INI 文件中没指定的 ReadFromIni 字段, 以 Modem Database 的默认记录赋值。需要 TmodemInfo 的完整定义, 参见 AddModem。

下面例子从文件 c:\windows\win.ini 中的 “Modem Configuration” 区读 Modem 记录。

```
DB.ReadFromIni(ModemData, 'Modem Configuration', 'C:\WINDOWS\WIN.INI');
```

7.3.2 TApdModemDialer 控件

TApdModemDialer 是个拨号引擎, 不断地拨一个电话号码, 直到一个连接已建立或一个特定的号码拨打尝试失败。当一个拨号尝试失败时, 在尝试再拨之前, 拨号引擎等待指定的一段时间。TApdModemDialer 控件属性如图 7.13 所示。

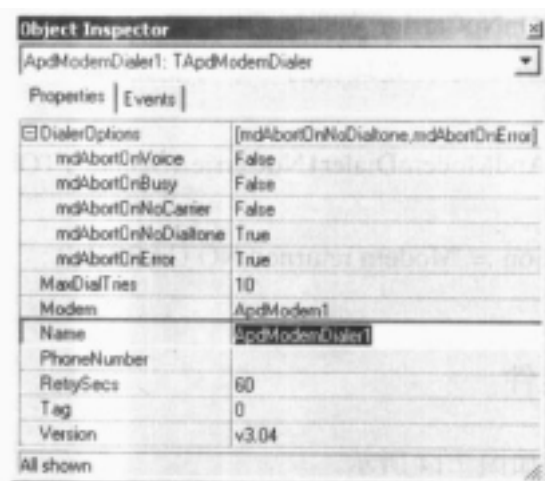


图 7.13 TApdModemDialer 控件属性

TApdModemDialer 不显示任何拨号或重试进度信息。它公布了相当多的事件，以提供给用户状态信息。实际上，Async Professional 包含了一个 TApdDialerDialog 控件，能在对话框中显示拨号进度信息。

1. OnDialStart 事件

OnDialStart 定义一个在拨号引擎拨一个电话号码前调用的事件处理器。

下面的例子显示 OnDialStart 事件的一个可能执行，它清空并复位在一个窗口中的拨号状态显示。

```
procedure Frm.ApdmModem1DialStart(Sender : TObject);
begin
  SecsRemainingLabel.Caption := IntToStr(ApdmModem1.DialTimeout);
  DialStatusLabel.Caption := 'Dialing';
end;
```

2. OnNoDialTone 事件

OnNoDialTone 定义一个 Modem 返回没有拨号音结果时调用的事件处理器。

下面例子显示 OnNoDialTone 事件处理器的一个可能执行。它更新窗口的 TLabel 控件以显示拨号器 (Dialer) 的状态。

```
procedure TMainForm.ApdmModemDialer1NoDialTone(Sender : TObject);
begin
  DialStatusLabel.Caption := 'Modem returned NO DIALTONE';
end;
```

3. OnNoCarrier 事件

OnNoDialTone 定义 Modem 返回 NO CARRIER 结果结果时调用的一个事件处理器。

OnNoCarrier 是 TApdModem 中的 OnModemNoCarrier 事件的镜像 (Mirror)。事实上，如果赋给拨号器 (Dialer) Modem 属性的 TApdModem 有 OnModemNoCarrier 事件，在

OnModemNoCarrier 之后，OnNoCarrier 立即被调用。

下面例子显示 OnNoCarrier 事件处理器的一个可能执行。它更新窗口的 TLabel 控件以显示拨号器 (Dialer) 的状态。

```
procedure TMainForm.ApdmModemDialer1NoCarrier(Sender : TObject);
begin
    DialStatusLabel.Caption := 'Modem returned NO CARRIER';
end;
```

7.3.3 TAdTerminal 控件

TAdTerminal 控件属性如图 7.14 所示。

TAdTerminal 控件代表终端的可视的部分。它是 Async Professional 终端类套件中仅有的可视控件。它负责维护窗口操作，并执行低级的处理，以获取在 PC 键盘上所有可能的击键 (Keystrokes)。

除了这个，它自己不执行任何工作，而是将大多数的显示和其他能力交给仿真器控件 (TAdTerminalEmulator 的继承者)。本质上，Terminal 控件在 PC 屏幕和键盘和其他执行所有工作的类型充作一个导管 (Conduit)。

为了终端窗口显示数据，有两个对象须连接到终端控件，即 COM 端口 (TApdComPort 控件) 和一个仿真器 (TAdTerminalEmulator 的下级控件)。为了能够使用 Terminal 控件，必须放一个 Terminal、一个 COM 端口和 Terminal 仿真器在窗体中，通过设置 COM 端口和仿真器属性将之互联。一旦它们在这个方式下连接，设置 Terminal 的 Active 属性为 True，就可以开始使用终端了。

COM 端口将来自串行通信设备 (串口或 Winsock 层) 的输入字节提供给 Terminal。而终端依次提供必须输出到串口设备的数据。就其本身而言，Terminal 控件并不知道如何处理输入的数据，它没有 Terminal 控制顺序 (Terminal Control Sequences) 和如何将文本分开的内在知识 (Built In Knowledge)。



图 7.14 TAdTerminal 控件属性

因此，它直接将输入数据流传给终端仿真器控件（Terminal Emulator Component）。而这个控件使用一个 Parser（分析器）对象来识别 Terminal 控制命令，一个 Buffer 对象来存储可显示的数据。这个终端也将所有键盘输入传给仿真器，以便让仿真器能处理按键和决定返回到主机的内容（它将用键盘映射对象（Keyboard Mapping Object）去做）。这个仿真器将用 Terminal 控件的 COM 串口通过串行设备返回数据给主机。

至于显示，仿真器将接管 Terminal 控件的窗口句柄并直接在画板里绘图。

Emulator 控件决定终端用的 Emulator 控件。

赋给这个属性的值或者是 Nil（其中没有执行仿真），或者是 TApdEmulator 类的正确初始化的实例。

在设计期间，Emulator（仿真器）常常自动设置为 TApdTerminal 在窗体（Form）中找到的第一个 Emulator 控件。如果需要，用对象观察器选择一个不同的 Emulator 控件。

仅当使用动态创建的 TApdEmulator 或在多个 TApdEmulator 控件进行选择时，在运行时间（Run Time）设置仿真器属性才是需要的。

7.3.4 TApdPhoneNumberSelector 控件

TApdPhoneNumberSelector 控件属性如图 7.15 所示。

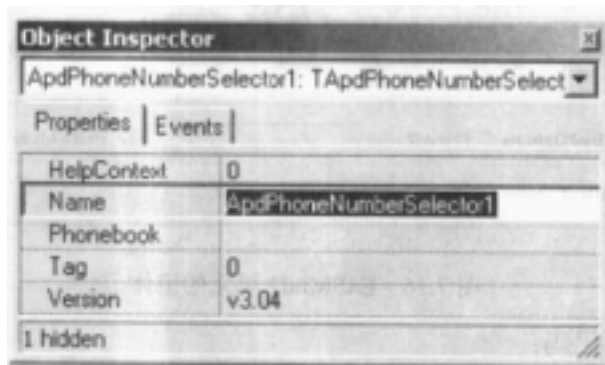


图 7.15 TApdPhoneNumberSelector 控件属性

TApdPhoneNumberSelector 对话框是个包含一个组合框（可输入电话号码）和一个选择按钮（用户能单击选择来自一个 Phonebook 数据库的电话号码）的模态对话框（Modal Dialog Box）。当调用 TApdPhoneNumberSelector.Execute 时，它临时创建 TgetNumberForm 窗体（包括一个 DelphiTcomboBox 和几个 TbitBtn 控件）的实例。

如果用户单击 Select 按钮，电话号码选择器（来自 AdSelNum 单元的 TnumberSelectForm 窗体）临时被创建以选择来自 Phonebook 数据库的电话号码。

在用户输入或选择一个号码并单击 OK 按钮后，读 SelectedNumber 属性可获得选中的电话号码字符串。

要能使用电话号码检出器，需简单创建一个实例，赋 TApdPhonebook 类的实例给 Phonebook 属性，执行这个对话框。

7.4 基于 APRO 组件的例子

下面给出了两个简单的例子。

1. 例 1

下面的例子为调用 TApdComPort、TApdSModem 和 TAdTerminal 控件进行拨号的程序。例子较简单，笔者不作详细解释。在命名为 ExSModem 的窗体中放置如图 7.16 中的控件，之后将单元文件命名为 ExSMod1.pas。

ExSMod1 单元的窗体如图 7.16 所示。

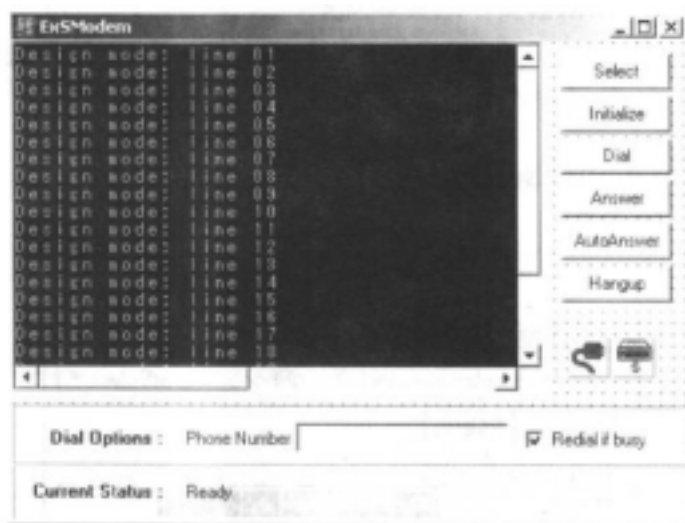


图 7.16 ExSMod1 单元的窗体

ExSMod1 单元的源代码:

```
unit ExSMod1;
```

```
interface
```

```
uses
```

```
WinTypes, WinProcs, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,  
StdCtrls, AdPort, AdSModem, ExtCtrls, OoMisc, ADTrmEmu;
```

```
type
```

```
TExSModem1 = class(TForm)
```

```
  ApdComPort1: TApdComPort;
```

```
  InitializeButton: TButton;
```

```
  AnswerButton: TButton;
```

```
  AutoAnswerButton: TButton;
```

```
  DialButton: TButton;
```

```
  SelectButton: TButton;
```

```

DialOptionsPanel: TPanel;
PhoneNumberEdit: TEdit;
PhoneNumberLabel: TLabel;
RedialCheckBox: TCheckBox;
Bevel1: TBevel;
DialOptionsLabel: TLabel;
CurrentStatusLabel: TLabel;
StatusMessage: TLabel;
HangupButton: TButton;
ApdSModem1: TApdSModem;
AdTerminal1: TAdTerminal;
procedure InitializeButtonClick(Sender: TObject);
procedure DialButtonClick(Sender: TObject);
procedure AnswerButtonClick(Sender: TObject);
procedure AutoAnswerButtonClick(Sender: TObject);
procedure SelectButtonClick(Sender: TObject);
procedure ApdSModem1.ConnectionStatus(ModemInstance: TObject;
    ModemStatus: TApdSModemStatusInfo);
procedure HangupButtonClick(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    ExSModem1: TExSModem1;

implementation

{$R *.DFM}

procedure TExSModem1.InitializeButtonClick(Sender: TObject);
begin
    ApdSModem1.Initialize;
end;

procedure TExSModem1.DialButtonClick(Sender: TObject);
begin

```

```
    ApdSModem1.PhoneNumber := PhoneNumberEdit.Text;
    ApdSModem1.Dial(RedialCheckBox.Checked);
end;

procedure TExSModem1.AnswerButtonClick(Sender: TObject);
begin
    ApdSModem1.Answer;
end;

procedure TExSModem1.AutoAnswerButtonClick(Sender: TObject);
begin
    ApdSModem1.AutoAnswer(2);
end;

procedure TExSModem1.SelectButtonClick(Sender: TObject);
begin
    ApdSModem1.SelectModem;
end;

procedure TExSModem1.ApdSModem1ConnectionStatus(ModemInstance: TObject;
    ModemStatus: TApdSModemStatusInfo);
begin
    StatusMessage.Caption := ModemStatus.GetModemStateMsg;
end;

procedure TExSModem1.HangupButtonClick(Sender: TObject);
begin
    ApdSModem1.Hangup;
end;

end.
```

2. 例 2

下面的例子为调用 TApdComPort 控件、TApdModem 控件和 TApdModemDialer 控件进行简单拨号的程序。在命名为 ExDialer 的窗体中放置如图 7.17 中的控件，之后将单元文件命名为 Exdiale0.pas。Exdiale0 单元的窗体如图 7.17 所示。

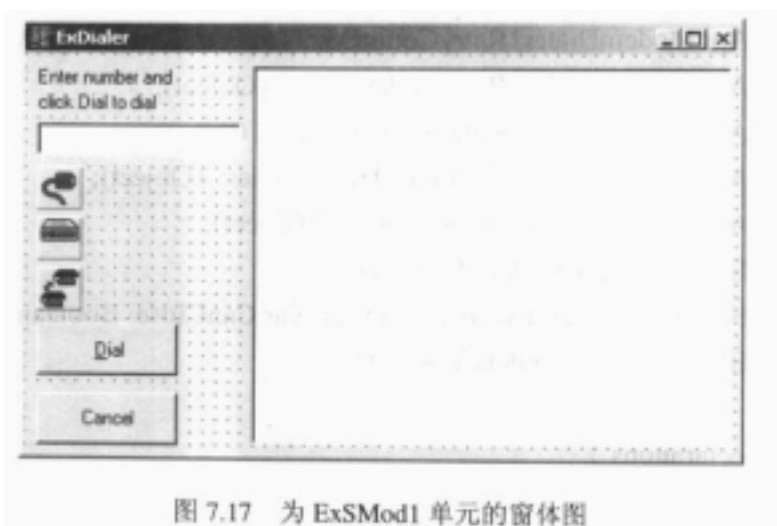


图 7.17 为 ExSMod1 单元的窗体图

```
unit Exdiale0;
```

```
interface
```

```
uses
```

```
    SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,  
    Forms, Dialogs, AdDial, AdModem, AdPort, StdCtrls, Buttons, OoMisc;
```

```
type
```

```
    TForm1 = class(TForm)
```

```
        ListBox1: TListBox;
```

```
        Label1: TLabel;
```

```
        Label2: TLabel;
```

```
        Edit1: TEdit;
```

```
        BitBtn1: TBitBtn;
```

```
        ApdComPort1: TApdComPort;
```

```
        ApdModem1: TApdModem;
```

```
        ApdModemDialer1: TApdModemDialer;
```

```
        Button1: TButton;
```

```
        procedure FormCreate(Sender: TObject);
```

```
        procedure ApdModemDialer1Busy(Sender: TObject);
```

```
        procedure ApdModemDialer1Connect(Sender: TObject);
```

```
        procedure ApdModemDialer1ConnectionEstablished(Sender: TObject);
```

```
        procedure ApdModemDialer1CycleDial(Sender: TObject);
```

```
        procedure ApdModemDialer1DialCount(M: TObject; Remaining: Word);
```

```
        procedure ApdModemDialer1DialStart(Sender: TObject);
```

```
        procedure ApdModemDialer1Error(Sender: TObject);
```

```
        procedure ApdModemDialer1GotLineSpeed(M: TObject; Speed: Longint);
```

```
        procedure ApdModemDialer1NoCarrier(Sender: TObject);
```

```
        procedure ApdModemDialer1NoDialTone(Sender: TObject);
```



```
    procedure ApdModemDialer1RetryCount(M: TObject; Remaining: Word);
    procedure ApdModemDialer1RetryEnd(Sender: TObject);
    procedure ApdModemDialer1RetryStart(Sender: TObject);
    procedure ApdModemDialer1TooManyTries(Sender: TObject);
    procedure ApdModemDialer1Voice(Sender: TObject);
    procedure BitBtn1Click(Sender: TObject);
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
    procedure Button1Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
    procedure AddStatusLine(const Msg : String);
end;

var
    Form1: TForm1;
implementation
{$R *.DFM}

procedure TForm1.AddStatusLine(const Msg : String);
begin
    Listbox1.Items.Add(Msg);
    Listbox1.ItemIndex := Pred(Listbox1.Items.Count);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    ApdComPort1.Open := True;
    ApdModem1.Started := True;
end;

procedure TForm1.ApdModemDialer1Busy(Sender: TObject);
begin
    AddStatusLine('Remote is busy');
end;

procedure TForm1.ApdModemDialer1Connect(Sender: TObject);
begin
```

```
AddStatusLine('Modem connected!');
end;

procedure TForm1.ApdmModemDialer1ConnectionEstablished(Sender: TObject);
begin
    AddStatusLine('Connection established!');
end;

procedure TForm1.ApdmModemDialer1CycleDial(Sender: TObject);
begin
    if ApdmModemDialer1.Retrying then
        AddStatusLine('Cancelling retry...')
    else
        AddStatusLine('Cycling dial attempt...');
end;

procedure TForm1.ApdmModemDialer1DialCount(M: TObject; Remaining: Word);
begin
    AddStatusLine('Dialing. ' + IntToStr(Remaining) + ' seconds to go');
end;

procedure TForm1.ApdmModemDialer1DialStart(Sender: TObject);
begin
    AddStatusLine('Starting a new dial attempt');
end;

procedure TForm1.ApdmModemDialer1Error(Sender: TObject);
begin
    AddStatusLine('Modem returned ERROR result');
end;

procedure TForm1.ApdmModemDialer1GotLineSpeed(M: TObject; Speed: Longint);
begin
    AddStatusLine('Connected at ' + IntToStr(Speed) + ' baud');
end;

procedure TForm1.ApdmModemDialer1NoCarrier(Sender: TObject);
begin
    AddStatusLine('Modem returned NO CARRIER');
```

end;

procedure TForm1.ApModemDialer1NoDialTone(Sender: TObject);

begin

 AddStatusLine('Modem returned NO DIALTONE');

end;

procedure TForm1.ApModemDialer1RetryCount(M: TObject; Remaining: Word);

begin

 AddStatusLine(IntToStr(Remaining) + ' seconds before new dial attempt');

end;

procedure TForm1.ApModemDialer1RetryEnd(Sender: TObject);

begin

 AddStatusLine('Retry finished. Will attempt to dial again.');

end;

procedure TForm1.ApModemDialer1RetryStart(Sender: TObject);

begin

 AddStatusLine('Dial failed. Will wait before retrying...');

end;

procedure TForm1.ApModemDialer1TooManyTries(Sender: TObject);

begin

 AddStatusLine('Too many failed dial attempts. Giving up.');

end;

procedure TForm1.ApModemDialer1Voice(Sender: TObject);

begin

 AddStatusLine('Remote answered with voice');

end;

procedure TForm1.BitBtn1Click(Sender: TObject);

begin

 if (Edit1.Text <> '') then begin

 ApModemDialer1.PhoneNumber := Edit1.Text;

 ApModemDialer1.Dial;

 end;

end;

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
    if ApdModemDialer1.Dialing then begin
        ApdModemDialer1.Abort;
        CanClose := True;
    end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    if ApdModemDialer1.Dialing then
        ApdModemDialer1.Abort;
end;

end.
```

本章小结

本章介绍了 SPComm 控件和 TurboPower 的 APRO 和 APRO 2.x 的部分通信控件的属性、方法和事件，并举了几个例子。TurboPower 公司的 APRO 组件为 Delphi 和 C++ Builder 开发者提供世界一流的通信工具套件。使用 APRO 组件，用户可以很快地开发功能很强的串口应用程序。

第 8 章 基于 MSComm 的多线程通信编程实例详解

本章主要内容:

- 系统简介
- 系统规划设计
- 源程序的分析

本章介绍一个系统实例,系笔者曾参与并负责的项目。项目的全称为程控机房的线路、配线架集中告警系统。本章具体介绍该系统中通信的实现。本章应用了诸多的 Delphi 技巧,如 ADO 的应用,请读者先熟练地掌握,笔者不作介绍了。

本章首先介绍系统的简介,然后是系统的规划设计,最后给出系统的通信部分的源码分析的详细说明。

8.1 系统简介

程控机房的线路、配线架集中告警系统由远端检测传输单元和中央监控中心组成。远端检测传输单元安装在配线架机房内,对 MDF 保安单元的告警状态、外线电缆断线等情况进行检测,当发现配线架告警或外线电缆断线时,通过传输单元(Modem)传输到中央监控中心,监控中心声光告警,并通过电话通知相关人员。同时,中央监控中心可以通过 Modem 对远端检测传输单元进行设置。系统的体系结构如图 8.1 所示。

系统由远端检测传输单元(或称为告警监测仪)和中央监控中心组成。

8.1.1 告警监测仪(包括监测单元、调制解调器、采集器)

告警监测仪特征如下:

- (1) 分布于各交换机分局,采集并传输数据。
- (2) 传输数据使用调制解调器。
- (3) 使用工控机,可以工作在 DOS 或 Windows NT 环境下。
- (4) 系统长期无人值守,软硬件稳定可靠。
- (5) 检测列单元告警信号,并声(光)告警。
- (6) 实时将告警信息,恢复信息传到监控中心。
- (7) 定时传数据,可从监控中心校参数。

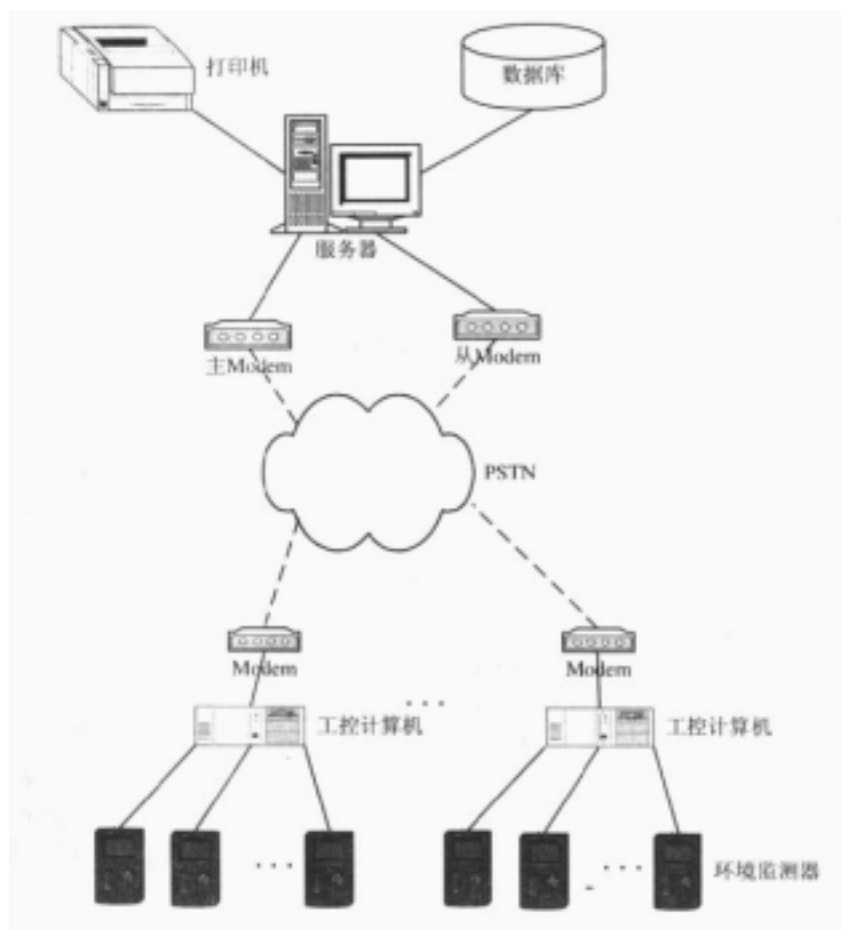


图 8.1 系统的体系结构

8.1.2 监控中心

监控中心特征如下：

- (1) 通过调制解调器控制所有监测单元，并采集数据。
- (2) 控制调制解调器数 ≥ 2 ，并工作于主，副备份方式。
- (3) 对传来数据进行存储，管理，并按类别进行告警和动态显示。
- (4) 控制“告警监测仪”数 ≥ 150 个。
- (5) 告警功能：接收告警监测仪告警信息，以图形，文字和声音形式告警。
- (6) 数据库管理功能：对告警信息、恢复信息、测试信息进行数据库处理，用户可通过操作界面进行查询，打印。
- (7) 自动测试功能：定时，随时对各分局的告警监测仪进行自动，人工测试，并显示，记录测试结果。
- (8) 图形功能：可显示所辖区域地图及下属分局配线架模拟图。

接下来讨论系统的规划和设计，详细说明系统的模块、通信协议的设计和系统的数据库的设计。

8.2 系统规划设计

监控中心功能模块图如图 8.2 所示, 监控中心功能模块分为四大模块: 通信模块、图形管理模块、常规数据库维护、监测单元控制模块。

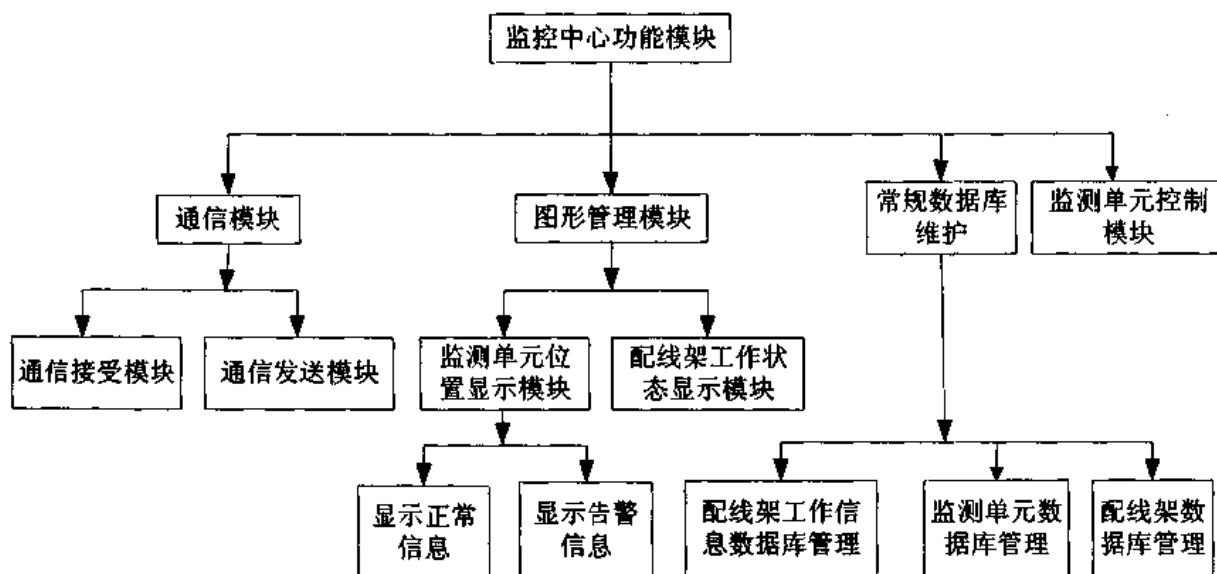


图 8.2 监控中心功能模块

8.2.1 各模块说明

下面详细说明各模块的构成和功能。

1. 监测单元模块控制

提供自动测试、人工测试能力。其中, 人工测试的优先级高于自动测试。即当进行人工测试时, 暂时停止进行自动测试, 当人工测试完成之后, 可以继续自动测试。监测单元功能模块图如图 8.3 所示。

自动测试模块和人工测试模块一般执行以下步骤。

- (1) 控制命令生成: 生成监测控制命令, 即要求测试单元采集信息并发送当前的工作状态。
- (2) 控制命令发送: 由监控中心发送控制命令。
- (3) 测试结果接收: 控制命令发送之后等待测试结果。当测试结果达到之后, 监控中心接收测试结果。
- (4) 测试结果存储: 存储测试结果到相应的数据库中。

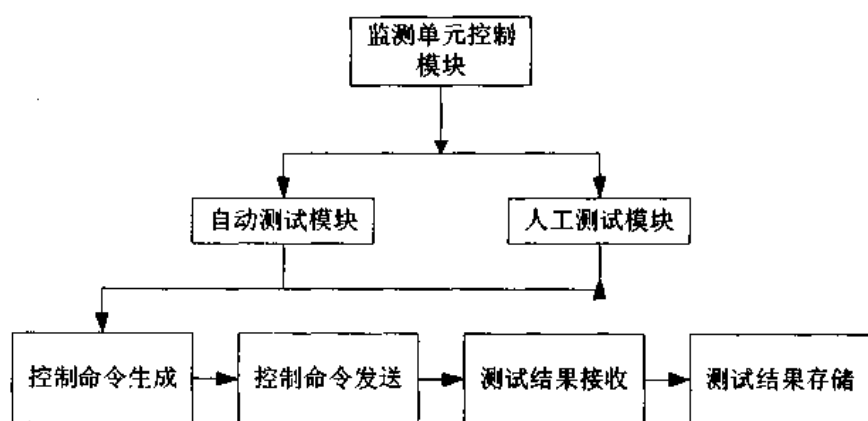


图 8.3 监测单元功能模块

2. 常规数据库维护模块

数据库管理功能：对告警信息、恢复信息、测试信息进行数据库处理，用户可通过操作界面进行查询，打印。

3. 图形管理模块

该模块分成监测单元位置显示模块、配线架工作状态显示模块两个模块。监测单元位置显示模块负责显示监测单元在地图上的具体位置和当前的状态。当前的状态分为正常状态和告警状态。监控中心定时刷新图形的显示。配线架工作状态显示模块负责显示配线架工作状态，当用户查询某一个监测单元时，将显示配线架工作状态。

根据配线架的工作状态信息，系统决定执行相应的告警动作。如果发现从配线架采集的信息不正常，则系统执行规定的告警动作。

表 8.1 告警信息分类

告警类型	告警方式	备注
通信不畅	黄灯	可能是通信的 Modem 出错，电话线中断等。一般错误
列架出错	闪烁的红灯	紧急错误
运行正常	绿灯	

4. 通信模块

通信模块包括以下 3 个子模块。

(1) 通信初始化模块：此模块为公共模块。通信接收模块和通信发送模块均含该模块。

对通信设备进行初始化，分别对串口和 Modem 进行初始化，如果发现串口或者 Modem 出错，向用户提示出错信息。由于本系统应用于电信部门，必须保证稳定可靠，所以在设计本模块时，应加入出错处理，以便处理诸如串口出错、Modem 无法响应、Modem 占线等异常情况。当碰到上述情况时，系统一方面继续初始化串口和 Modem，另一方面向用户提示出错信息，以使用户能发现错误，并采取相应的措施。

(2) 通信接收模块：接收来自于监测单元的信息，并将监测单元的信息存储于相应的数

数据库中，并进行适当的显示。

(3) 通信发送模块：发送信息给监测单元。监控中心通过该模块发送控制命令到监测单元。

8.2.2 通信协议

1. 通信数据包的一般格式

通信数据包在本章中，又称之为通信信息包。其格式如图 8.4 所示。

发送方的格式说明。发送的信息包包括以下内容：

命令代码，指定信息包的作用；

信息报长度，整个信息包的长度值，其值为 64 Byte；

正文；

校验码，长度为 2 Byte。

接收方的格式说明。接收方接收到发送方的信息包后，首先校验该信息包是否正确，如果信息包校验失败，即信息包传送出错，则接收方抛弃该信息包，并断开 Modem 建立的通信链路，发送方等待接收方的反馈信息，如果在指定的时间未接收到反馈信息，发送方断开 Modem 建立的通信链路，发送方等待一定的时间，重发信息包。如果信息包校验正确，则接收方将正确的信息包发给发送方作为反馈信息，并对信息包进行相应的处理，发送方接收到反馈信息，进行校验。如果与发送的信息包相同，则表示信息包发送成功，否则需要重发信息包。

采用定长报文，报文长度定为 64Byte

通信数据包的一般格式（单位为Byte）：

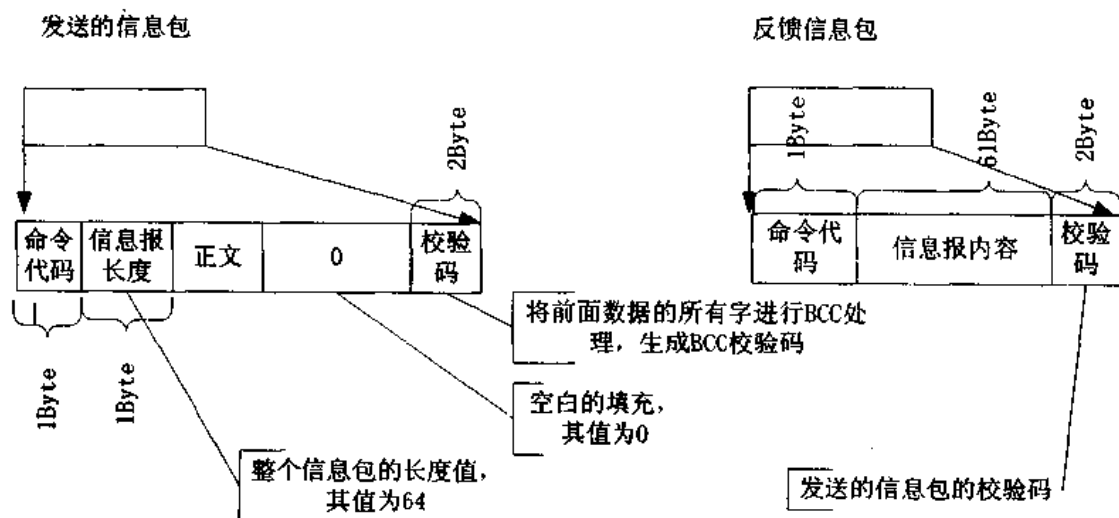


图 8.4 通信数据包的一般格式

所以反馈信息包的结构和内容与发送信息包的相同。

2. 监测单元到监控中心的信息包

监测单元到监控中心的信息包有两种,其一为包含被监测设备的工作状态信息的信息包,其二为被监测设备的配置信息包。信息包和反馈信息包的结构如图 8.5 所示。

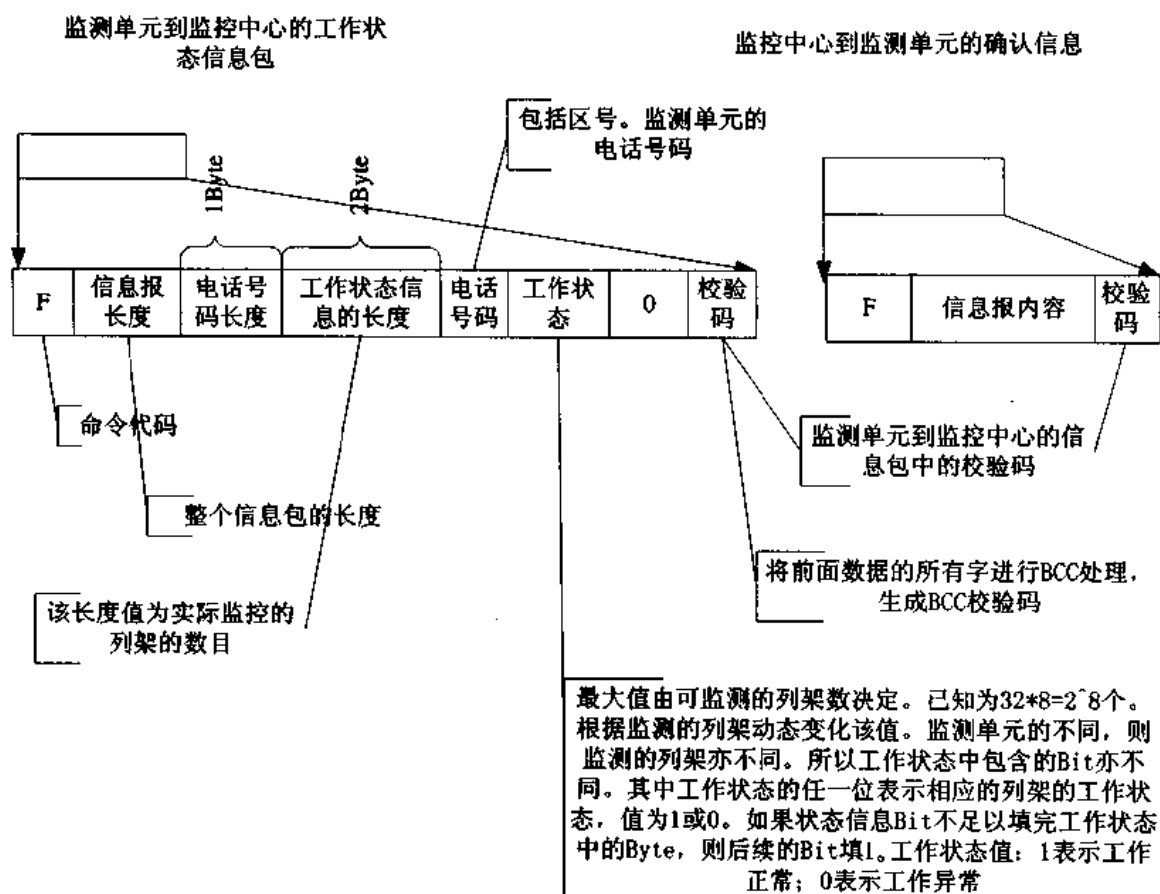


图 8.5 包含被监测设备的工作状态信息的信息包

信息包 1 为被监测设备的工作状态信息。其中的内容如下。

- 命令代码：规定为 F。
- 信息报长度：为指定大小，系统规定为 64 Byte。
- 电话号码的长度：考虑到以后电话号码的长度可能变化，比如地区电话号码升级，所以监测中心要求监测单元发送的信息报中包含有电话号码的长度。
- 监测单元的电话号码：监控中心根据监测单元的电话号码确定监测单元的位置，监控中心根据信息报中的电话号码的长度，解码电话号码。
- 被监测设备的工作状态信息的长度：该长度值为实际监控列架的数目。
- 被监测设备的工作状态信息：最大值由可监测的列架数决定。已知为 $32 \times 8 = 2^8$ 个。根据监测的列架动态变化该值。监测单元的不同，则监测的列架亦不同。所以工作状态中包含的 Bit 亦不同。其中工作状态的任一位表示相应的列架的工作状态，值为 1 或 0。如果状态信息 Bit 不足以填完工作状态中的 Byte，则后续的 Bit 填 1。工作状态值：1 表示工作正常，0 表示工作异常。

□ 校验码：监测单元到监控中心的信息包中的校验码。下面会讨论校验码的生成。

接收方（这里指监控中心）接收到信息包，校验无误后，发送反馈信息包给监测单元。其中，反馈信息包的结构和内容与发送信息包的相同。接收方将信息包解码，进行必要的处理，一方面将信息反馈到程序的主界面，另一方面将解码的信息存入到数据库中。

信息包 2 为被监测设备的配置信息包。信息包 2 的结构和内容基本与信息包 1 的相同，只有如下的几个不同。

□ 配置信息的长度：该长度值等于实际监控的列架的数目。

□ 配置信息：最大值由可监测的列架数决定。已知为 $32 \times 8 = 2^8$ 个。根据监测的列架动态变化该值。监测单元的不同，则监测的列架亦不同。配置信息第一次来自于监测单元，由监测单元负责解释执行。其后用户可以通过在监测中心配置该配置信息包，然后发送给监测单元，从而实现远程控制监测单元的信息解释。其中 1 表示异常，0 表示正常。

信息包和反馈信息包的结构如图 8.6 所示。

监测单元到监控中心（或监控中心到监测单元）的配置信息包

监控中心到监测单元（或监控中心到监测单元）的确认信息

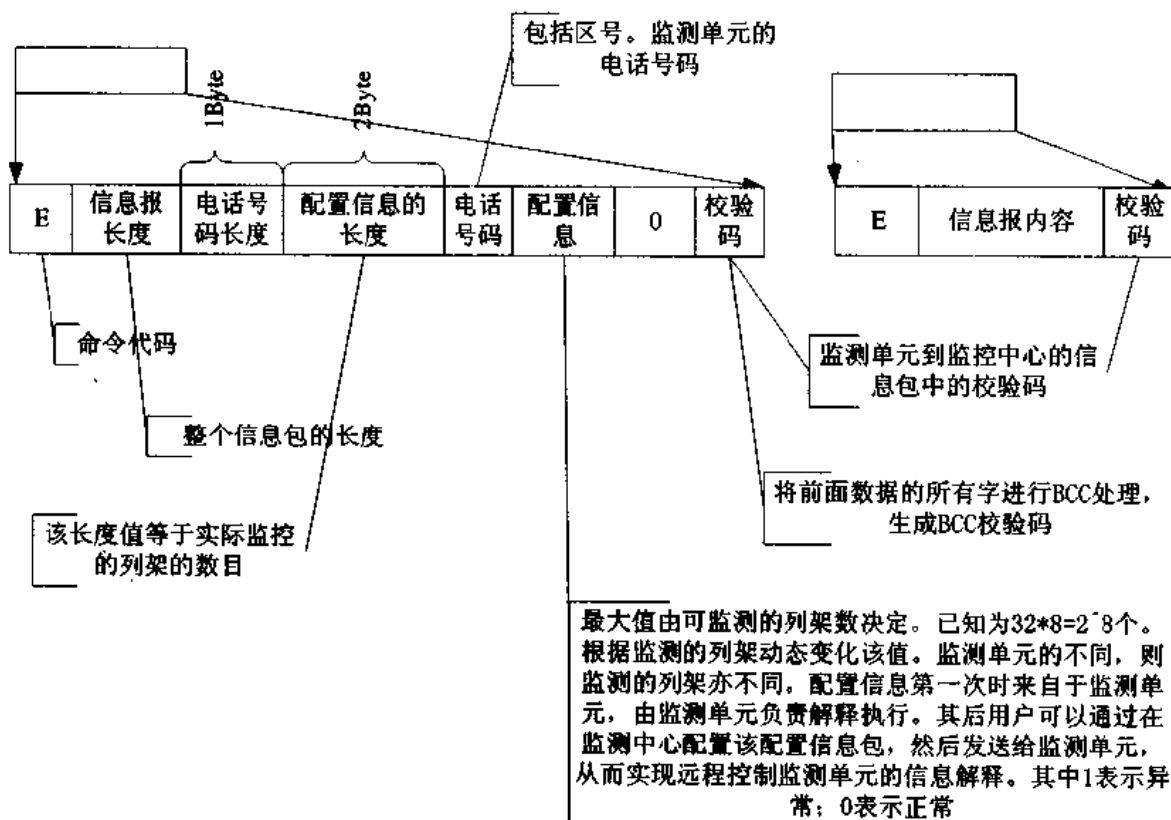


图 8.6 被监测设备的配置信息包

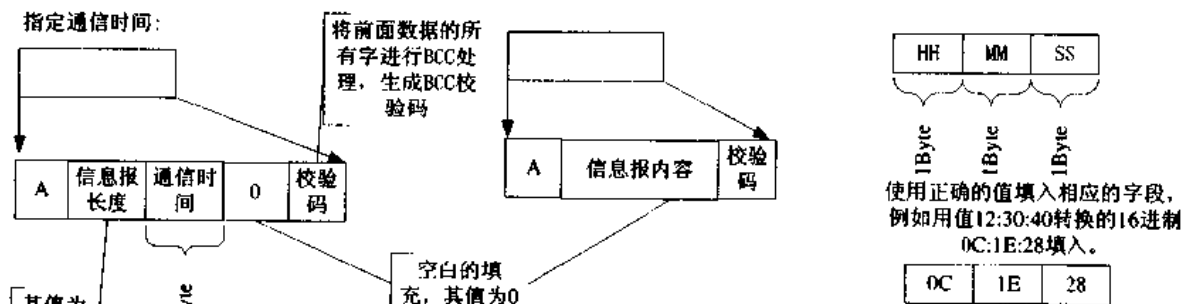
3. 监控中心到监测单元的信息包

监控中心到监测单元的信息包如图 8.7 所示。

监控中心到监测单元的信息包

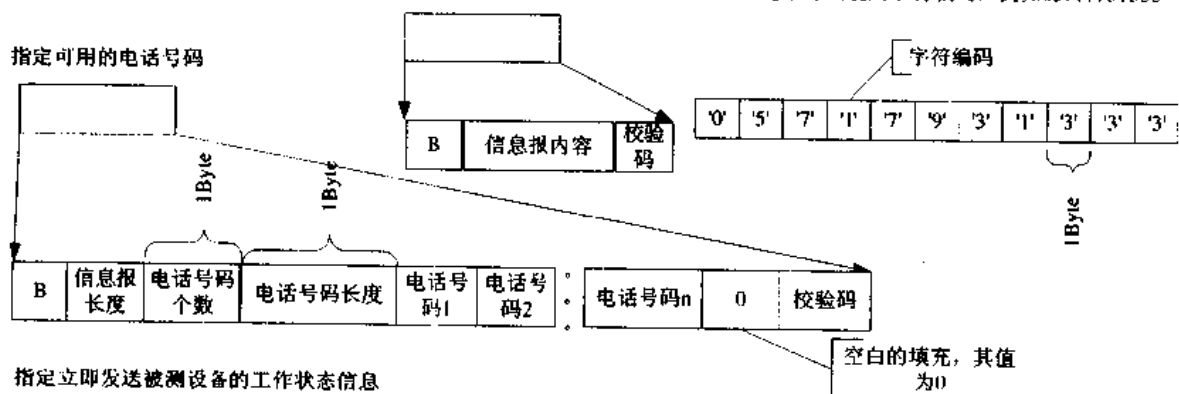
监测单元到监控中心的确认信息

通信时间格式: HH:MM:SS



电话号码使用字符编码: 例如05717931333

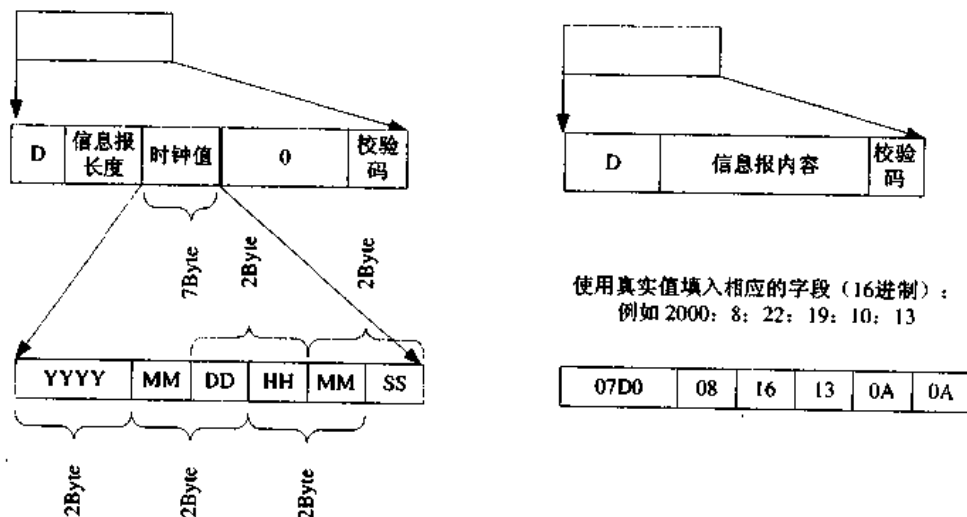
指定可用的电话号码



指定立即发送被测设备的工作状态信息



校正时钟



时钟值的格式为: YYYY:MM:DD:HH:MM:SS

采用定长报文, 报文长度定为64Byte

图 8.7 监控中心到监测单元的信息包

监控中心到监测单元的信息包分为 5 个。

□ 指定通信时间: 命令代码为 A, 其中通信时间格式为 HH:MM:SS, 要求用正确的值填入相应的字段, 例如用值 12:30:40 转换的 16 进制 0C:1E:28 填入。

□ 指定可用的电话号码：命令代码为 B，用于通知监测单元监控中心的可用电话号码。可用电话号码按主用电话号码、备用电话号码的顺序填写在该信息包中。监测单元在与监控中心通信时，首先使用主要电话号码与监控中心通信，如果发现监控中心的主要电话忙，则试图使用备用电话号码与监控中心通信。

□ 指定立即发送被测设备的工作状态信息：命令代码为 C，用户通过该命令可以要求指定的监测单元立即发送被测设备的工作状态信息。由于 Modem 主要用于接收监测单元的信息包，所以这个命令通过备用 Modem 发送给指定的监测单元。

□ **校正时钟**：命令代码为 D，考虑到监测单元的时钟可能与监控中心的时钟不一致，因而不能在监控中心指定的时间发送被测设备的工作状态信息，所以使用监控中心的时钟校正监测单元的时钟。

□ 设定监测单元的配置信息：命令代码为 E，用户可以通过监控中心配置监测单元的配置信息。

8.2.3 通信日志设计

通信日志用于提示用户当前系统的工作状态。通信日志分两种：一个为监控中心的通信日志，另一个为监测单元的通信日志。下面分别给出介绍。

1. 监控中心的通信日志

记录整个通信过程。一般格式为：时间+执行动作或者时间+与执行相关的内容。比如：时间+执行动作，

2000-10-13 05:00:54--串口 2:接收线程: 开始接收来自于监测单元的状态信息包

时间+与执行相关的内容,

2000-10-13 05:00:55--串口 2:接收线程: 接收到的信息包为:

[illegible]

一个完整的接收过程为：

2000-10-13 04:01:03--串口 2: Modem 初始化成功!

2000-10-13 05:00:54--串口 2:接收线程: 开始接收来自于监测单元的状态信息包

2000-10-13 05:00:55--串口 2:接收线程: 接收到的信息包为:

[illegible]

2000-10-13 05:00:55--串口 2:接收线程: 发送 ACK 数据包给监测单元!

2000-10-13 05:00:55--串口 2: 10 进制: 70 64 3 0 150 56 48 52 254 255 255 255 255 254

255 255 255 255 254 255 255 255 255 254 255 255 255 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

000000000000000000000000 148 208

2000-10-13 05:00:55--信息包来自于串口 2

信息包内容为

监测单元的电话号码为： 804

监测单元的列架数为: 150

监测单元的坏单元数目为: 4

2000-10-13 05:00:55--串口 2:接收线程:接收的数据已经存入到数据库中!

2000-10-13 05:00:56--串口 2: 断开与监测单元的连接并挂机!

一个完整的发送过程为:

2000-10-10 12:52:51--串口 1:发送线程: 第 1 次呼叫!

2000-10-10 12:53:15--串口 1:发送线程: 数据链路建立!

2000-10-10 12:53:15--串口 1:发送线程: 第 1 次发送!

2000-10-10 12:53:15--串口 1:发送线程: 开始接收 ACK 数据包!

2000-10-10 12:53:17--串口 1:发送线程: 接收到的信息包:

2000-10-10 12:53:17--串口 1: 10 进制: 69 64 3 0 32 56 48 49 0 2 8 0 0 0 0 0 0

[illegible]

2000-10-10 12:53:17--串口 1:发送线程: 接收到的 ACK 信息包正确!

2000-10-10 12:53:17--串口 1:发送线程: 接收到正确的 ACK 数据包!

2000-10-10 12:53:17--串口 1: 断开与监测单元的连接并挂机!

2000-10-10 12:53:19--串口 1:发送线程; 终止发送线程!

2000-10-10 12:53:19--串口 1: 断开与监测单元的连接并挂机!

2000-10-10 12:53:28--串口 1: Modem 初始化成功!

2. 监测单元的通信日志

记录整个通信过程。一般格式为：时间+执行动作或者时间+与执行相关的内容。比如：时间+执行动作。记录内容与监控中心相同，不过语言使用英文。

8.2.4 数据库设计

数据库设计用于保存监测单元的信息和状态信息, 以及监控中心的一些信息。下面给出各数据库的详细的設計。数据库表单的关系图如图 8.8 所示。

1. 表名: AlertInfo

表 8.2 存储告警信息。当监控中心接收到数据包后，接收线程分析该数据包，如果数据包为监测单元的被监测设备的状态信息包，并且该数据包表明被监测设备工作异常，则接收线程将告警信息写入此表。该表用于为系统提供语音扩展功能，当安装好语音服务器后，运行于语音服务器的软件监视此表，当发现异常时，利用本表向用户提供语音服务，向用户发出语音告警。

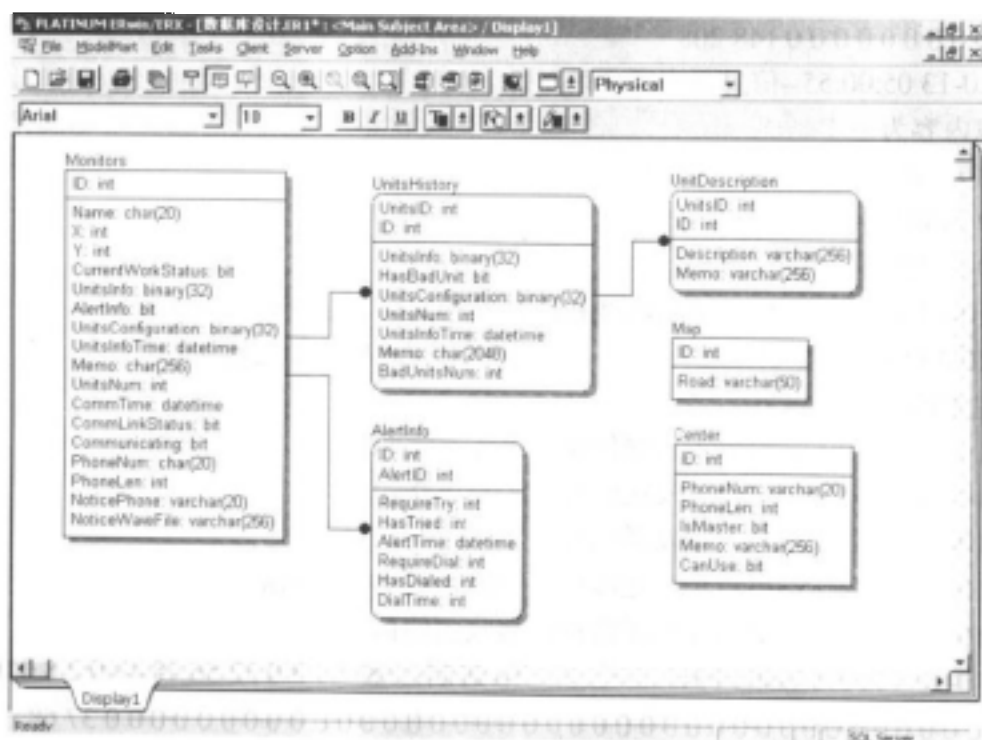


图 8.8 数据库表单的关系图

注意: 数据类型为空白表明该表列名对应的数据类型与上一行的表列名对应的数据类型相同。本数据库设计系基于 SQL Server 7.0, 所以读者将发现数据类型中会有 SQL Server 7.0 的特有类型。

表 8.2

告警信息

表列名	表列名的数据类型	表列名的备注	表列名是主键
ID	int	监测单元的编号	Yes
AlertID	int	告警信息编号, 自增加, 步长为 1	Yes
RequireTry	int	要求尝试次数。缺省值为 3	No
HasTried	int	已经尝试次数。缺省值为 0	No
AlertTime	datetime	告警时间, 缺省值为当前时间	No
RequireDial	int	要求拨通次数, 缺省值为 1	No
HasDialed	int	已经拨通次数, 缺省值为 0	No
DialTime	int	当前拨打时间	No

2. 表名: Center

表 8.3 存储监控中心的一些数据。其中的可用电话号码在系统初始化时由用户发往监测单元。

表 8.3 监控中心可用电话号码数据库, 通信设置信息库

表列名	表列名的数据类型	表列名的备注	表列名是主键
ID	int	编号, 自增加, 步长为 1	Yes
PhoneNum	varchar(20)	可用的电话号码	No
PhoneLen	int	电话号码的长度	No
IsMaster	bit	是否为主机, 1 表示主用 0 表示备用 (缺省值)	No
Memo	varchar(256)	备注	No
CanUse	bit	1 表示该电话号码可用了 0 表示该电话号码不可用	No

3. 表名: Map

表 8.4 用于存储地图信息, 即地图文件所在的路径, 监控中心的图形管理模块通过该路径找到地图文件。

表 8.4 地图信息

表列名	表列名的数据类型	表列名的备注	表列名是主键
ID	int	编号, 自增加, 步长为 1	Yes
Road	varchar(50)	地图文件的位置	No

4. 表名: Monitors

监测单元的相关信息。该表中存有监测单元的详细的消息, 包括地理坐标、对应的电话号码。接收线程接收到监测单元的数据包, 首先判断数据包是否正确, 如果正确, 则解析数据包。如果数据包为配置信息包, 将配置信息存入到对应位置。如果数据包为被监测设备的状态信息, 解开该数据包, 并分析相应的字段, 从中抽取出告警信息, 并根据结果进行分析。如果接收的数据包表明被监测设备出现异常, 将分析结果写入到 AlertInfo 表中。不管结果如何, 均将结果存入到表 8.5 中和表 8.7 中。

表 8.5 监测单元的详细信息

表列名	表列名的数据类型	表列名的备注	表列名是主键
ID	int	监测单元的编号, 自增加	Yes
Name	char(20)	监测单元所在地方的分局名称	No
X	int	地理坐标	No
Y	int	地理坐标	No

续表

表列名	表列名的数据类型	表列名的备注	表列名是主键
CurrentWorkStatus	bit	当前的工作状态: 0 为人工测试; 1 为自动测试 (缺省值)	No
UnitsInfo	char(32)	列架的详细信息, 显示列架的当前状态: 1 表示工作正常 (缺省值); 0 表示工作异常。 读写规则: 从高到低读写	No
AlertInfo	bit	告警信息: 0 表示运行正常; 1 表示列架出错 (缺省值) 根据 UnitsInfo 而获得的信息	No
UnitsConfiguration	char(32)	第一次的配置信息来自监测单元, 由单元负责解释执行。其后用户可以通过在监测中心配置信息包, 然后发送给单元, 从而实现远程控制监测单元的信息解释	No
UnitsInfoTime	datetime	最后一次获得列架信息的时间	No
Memo	char(256)	备注信息	No
UnitsNum	int	该分局的列架个数 (缺省值为 0)	No
CommTime	datetime	监测单元与检控中心的通信时间, 由监控中心负责分配通信时间	No
CommLinkStatus	bit	通信链路的状况: 1 表示工作正常 (缺省值); 0 表示工作异常	No
Communicating	bit	表示是否正在通信 1 表示正在与该监测单元通信; 0 表示现在没有与该监测单元通信 (缺省值)	No
PhoneNum	char(20)	监测单元使用的电话号码, 编码规范: 区号+电话号码, 如 05717931600	No
PhoneLen	Int	电话号码的长度	No
NoticePhone	varchar(20)	通知号码	No
NoticeWaveFile	varchar(25)	通知语音文件	No

5. 表名: UnitDescription

表 8.6 提供对监视的列架的描述信息。

表 8.6 列架的描述信息

表列名	表列名的数据类型	表列名的备注	表列名是主键
UnitsID	int	列架信息的编号	Yes
ID	int	监测单元的编号	Yes
Description	varchar(256)	列架的描述信息	No
Memo	varchar(256)	备注	No

6. 表名: UnitsHistory

用户可以在表 8.7 中查询监测单元发送的信息。监控中心的图形管理模块显示某个监测单元某一时刻的被监控设备的状态信息时,首先读取该时刻相应的配置信息,根据配置信息,对被监控设备的状态信息进行翻译,然后显示出来。同一行中的配置信息和状态信息表示该时刻被监控设备的配置信息和当时的状态信息。

表 8.7 列架的详细历史信息

表列名	表列名的数据类型	表列名的备注	表列名是主键
UnitsID	int	列架信息的编号, 自增加	Yes
ID	int	监测单元的编号	Yes
UnitsInfo	char(32)	列架的详细信息, 显示列架的当前状态: 1 表示工作正常 (缺省值); 0 表示工作异常	No
HasBadUnit	bit	列架中是否有单元工作异常, 其值等效于 Monitors 中的 AlertInfo 1 表示工作正常 (缺省值); 0 表示工作异常	No
UnitsConfiguration	char(32)	配置信息第一次时来自于监测单元, 由监测单元负责解释执行。其后用户可以通过在监测中心配置信息包, 然后发送给监测单元, 从而实现远程控制监测单元的信息解释	No
UnitsNum	int	列架的个数 (缺省值为 0)	No
UnitsInfoTime	datetime	获得列架信息的时间	No
Memo	char(2048)	备注	No
BadUnitsNum	int	(列架中) 工作异常的单元的数目 (缺省值为 0)	No

8.3 源程序的分析

考虑到监控中心的源码基本与监测单元的源码相同。而监控中心的设计要远复杂于监测单元的设计。所以，笔者主要分析监控中心的源码。另外，本书主要集中讨论通信编程，所以笔者将分析的重点放在监控中心的通信模块。为了将通信模块分析得更详尽，笔者将讨论相关的模块，并逐步地分析代码。

8.3.1 循环冗余校验 CRC 算法源程序的分析

在远距离数据通信中，为确保高效而无差错地传送数据，必须对数据进行校验即差错控制。传统的差错检测法有：奇偶校验、校验和检测法、行列冗余码校验法、反向循环码等，上述方法都是增加信息的冗余量，与信息位同时发送出去，在接收端通过对数据信息进行比较、判别或简单的求和运算，然后将所得结果同接收的冗余位比较，若二者相同则认为接收正确，否则就判定有误码出现。然而这些方法仅仅能反映数据信息行、列奇偶情况，漏判的概率较高。

循环冗余校验 CRC (Cyclic Redundancy Check) 对一个传送数据块进行校验，是一种高效的差错控制方法。CRC 码校验由分组线性码的分支而来，其应用主要为二进制码组，所有码字的运算是封闭的，其误判的概率很低。

1. 循环冗余校验码原理

下面简述一下 CRC 校验方法的原理。

CRC 校验采用多项式编码方法，如一个 8 位二进制数 ($B_7B_6B_5B_4B_3B_2B_1B_0$) 可以用 7 阶二进制码多项式 $B_7X^7+B_6X^6+B_5X^5+B_4X^4+B_3X^3+B_2X^2+B_1X^1+B_0X^0$ 表示。

例如 11000001 可表示为：

$$1X^7+1X^6+0X^5+0X^4+0X^3+0X^2+0X^1+0X^0$$

一般说， n 位二进制数可用 $(n-1)$ 阶多项式表示。它把要发送的数据位串看成是系数只能为“1”或“0”的多项式。一个 n 位的数据块可以看成是从 X_{n-1} 到 X_0 的 n 项多项式的系数序列，位于数据块左边的最高位是 X_{n-1} 项的系数，次高位是 X_{n-2} 项的系数，以此类推，位于数据块右边的最低位是 X_0 项的系数，这个多项式的阶数为 $n-1$ 。

多项式乘除法运算过程与普通代数多项式的乘除法相同。多项式的加减法运算以 2 为模，加减时不进、错位，如同逻辑异或运算。

采用 CRC 校验时，发送方和接收方事先约定一个生成多项式 $G(X)$ ，并且 $G(X)$ 的最高项和最低项的系数必须为 1。设 m 位数据块的多项式为 $M(X)$ ，生成多项式 $G(X)$ 的阶数必须比 $M(X)$ 的阶数低。CRC 校验码的检错原理是：发送方先为数据块生成 CRC 校验码，使这个 CRC 校验码的多项式能被 $G(X)$ 除尽，实际发送此 CRC 校验码；接收方用收到的 CRC 校验码除以 $G(X)$ ，如果能除尽，表明传输正确，否则，表示有传输错误，请求重发。

生成数据块的 CRC 校验码的方法是：

(1) 设 $G(X)$ 为 r 阶，在数据块末尾添加 r 个 0，使数据块为 $m+r$ 位，则相应的多项式为 $X^rM(X)$ 。

(2) 以 2 为模, 用对应于 $G(X)$ 的位串去除对应于 $X^iM(X)$ 的位串, 求得余数位串。

(3) 以 2 为模, 从对应于 $X^iM(X)$ 的位串中减去余数位串, 结果就是为数据块生成的带足够校验信息的 CRC 校验码位串。

例如, 设要发送的数据为 1101011011, $G(X)=X^4+X+1$, 则首先在发送数据块的末尾加 4 个 0, 得到 11010110110000, 然后用 $G(X)$ 的位串 10011 去除, 再用 11010110110000 减去余数位串 1110, 得到的即为 CRC 位串 11010110111110, 将对应多项式称为 $T(X)$, 显然, $T(X)$ 能被 $G(X)$ 除尽。这样, 一旦接收到的 CRC 位串不能被同样的 $G(X)$ 的位串除尽, 那么一定有传输错误。

当使用 CRC 校验码进行差错控制时, 除了为 $G(X)$ 的整数倍的差错多项式不能被检测外, 其他差错均能被查出。CRC 校验码的差错控制效果取决于 $G(X)$ 的阶数, 阶数越高, 效果越好。目前, 常用的有两种生成多项式 $G(X)$ 的方法, 分别是:

CRC-16 $X^{16}+X^{15}+X^2+1$

CCITT $X^{16}+X^{12}+X^5+1$

CRC 校验码实际上是一种线性码, 将任意 CRC 校验码循环移位后仍然是一个 CRC 校验码。由于它有良好的结构, 检错能力强, 易于实现硬件编、译码, 因此在数据通信系统中得到广泛的应用。

2. CRC 校验码

对于某些不宜用硬件实现 CRC 校验而又需要用 CRC 校验码进行差错控制的系统中, 需用软件方法实现 CRC 校验, 即实现编码、检错和译码功能。

从 CRC 校验码编码规则可以看出, CRC 校验码实际上是由原始数据位串和紧跟其后的与 $G(X)$ 位串等长的冗余位串组成, 只要求出此冗余位串, 发送方即可将原始数据和冗余位串装配成一 CRC 位串序列后再发送。CRC 校验码译码非常简单, 只需从接收到正确 CRC 校验码尾部截掉与 $G(X)$ 位串等长冗余位串, 余下的部分即为原始数据位串。CRC 校验码错误检测按模 2 除法运算, 用接收到的 CRC 位串除以 $G(X)$ 位串, 看是否能够除尽即可确定。

3. 分析生成 CRC 信息包的程序

本函数中使用了自定义的类型, 说明如下:

type

tByteArray=array [1..cnPackageLen] of byte;

下面的 Delphi 语言模块生成信息包的 CRC 代码。

procedure GenerateCRC(var aPackage:tByteArray;const iLen:integer)为求 CRC 校验码子程序

说明: CRC 校验码必须为无符号型, 长度为 16 位。

参数 aPackage 为信息块数组, 字节型。

参数 iLen 为信息块所占字节数, 包括校验码占的字节数。

返回值为所求校验码。

目的: 生成信息包的 CRC 代码。

输入参数: aPackage: 信息包

ILen: 该信息包长度。

结果: 在信息包的末尾加入生成的 CRC 代码。

注意: 发送端调用该函数之前, 必须将校验码所占的两个字节置 0。接收端不必。
代码如下。

```

procedure GenerateCRC(var aPackage:tByteArray;const iLen:integer);
var
    crc:Word;
    ia,ib,ie,i,iCount:Integer;
    caTemp:tByteArray;
begin
    aPackage[cnPackageLen-1]:=0;
    aPackage[cnPackageLen]:=0;
    crc:=0;
    ia:=0;ib:=0;
    for i:=1 to iLen do
        caTemp[i]:=aPackage[i];
    ie:=caTemp[1];
    crc:=crc xor (ie shl 8);
    crc:=crc xor caTemp[2];
    iCount:=iLen;

    for i:=1 to (iLen-2)*8 do
        begin
            while iCount>2 do
                begin
                    if (caTemp[iCount] and 128)<>0 then
                        ia:=1
                    else
                        ia:=0;
                    caTemp[iCount]:=caTemp[iCount]shl 1;
                    if ib=1 then
                        caTemp[iCount]:=caTemp[iCount]+1;
                    ib:=ia;
                    iCount:=iCount-1;
                end;
            if (crc and 32768)<>0 then
                begin
                    crc:=crc shl 1;
                    if ia=1 then

```

```

        crc:=crc+1;
        crc:=crc xor 4129;
    end
else
    begin
        crc:=crc shl 1;
        if ia=1 then
            crc:=crc+1;
        end;
        iCount:=iLen;
    end;
    //GenerateCRC:=crc;
    aPackage[cnPackageLen-1]:=Hi(crc);
    aPackage[cnPackageLen]:=Lo(crc);
end;

```

4. 分析检验信息包的CRC程序

function CheckCRC(const aPackageReceived:trPackageReceived):Boolean 为校验信息包的CRC的函数程序。说明如下。

目的： 校验信息包的CRC。

输入参数：aPackageReceived 收到的信息包。

返回值：检查收到的信息包中的CRC代码，如果生成的CRC与接收的CRC相等，则返回True，否则返回False。

```

function CheckCRC(const aPackageReceived:trPackageReceived):Boolean;
var iOldCRCHi,iOldCRCLo:integer; //分别代表旧的CRC(16位)的高8位、低8位
    iNewCRCHi,iNewCRCLo:integer; //分别代表新的CRC的高8位、低8位
    anOriginalPackage:tByteArray;
    i:integer;
begin
    iOldCRCHi:=aPackageReceived.PackageGeneral[cnPackageLen-1];
    //获取接收信息包中的CRC代码
    iOldCRCLo:=aPackageReceived.PackageGeneral[cnPackageLen];
    //procedure GenerateCRC(var aPackage:tByteArray;const iLen:integer);
    for i:=1 to cnPackageLen-2 do
    //拷贝 aPackageReceived.PackageGeneral 到 anOriginalPackage 中
    begin
        anOriginalPackage[i]:=aPackageReceived.PackageGeneral[i];
    end;
    GenerateCRC(anOriginalPackage,CnPackageLen);

```

```

iNewCRCHi:=anOriginalPackage[cnPackageLen-1];
iNewCRCLo:=anOriginalPackage[cnPackageLen];
//将旧的 CRC 代码与新生成的 CRC 代码比较, 如果相等表示该数据包正确
//否则错误
if (iOldCRCHi=iNewCRCHi) and (iOldCRCLo=iNewCRCLo) then
    Result:=True
else
    result:=False;
end;

```

8.3.2 信息包的处理

首先介绍一下相关的常量定义和自定义类型。

//常量

```

const iArrayBase=0;    //在数组的基本指针
const iCRCNum    =2; //CRC 码的数目
const cnPhonesNum=10; //在监控中心使用的电话号码数目
const cnPhoneLen=20;  //电话号码长度
const cnBlank=0;      //填入到信息包的空白值,
const cnPackageLen=64; //信息报长度
const cnNormalUnitFlag=255; //0xFF 表示单元工作正常
const cnMaxNumStatusBytes=32; //工作状态字节数的最大的数目
const cnCommunicationTime=1/1440; //通信时间: 从初始化 Modem 开始, 拨号到
//远端的机器, 远端的机器接收到数据包, 并对包解析, 执行响应的动作的整个过程的
//时间, 用于校正发送给监测单元的系统时间

```

type

```

tByteArray=array [1..cnPackageLen] of byte;
tPhonesArray=array [1..cnPhonesNum] of string[cnPhoneLen];

```

type

```

trAvailablePhones    =record //可用电话
    aPhones:tPhonesArray; //可用电话数组, 第一个为主用电话, 其后的为备用电话
    iPhoneLen:integer;    //电话号码的长度
    iPhonesNum:integer;   //在监控中心使用的电话号码数目
end;
trPackageSetCommTime=record //指定通信时间的信息包
    PackageSetCommTime:tByteArray;
    iLen:integer;          //信息包的长度
end;
trPackageSetPhones    =record //指定可用的电话号码信息包

```

```

    PackageSetPhones :tByteArray;
    iLen:integer;
end;
trPackageSetDateTime=record //校正时钟的信息包
    PackageSetDateTime:tByteArray;
    iLen:integer;
end;
trPackageGetInfo =record //指定立即发送被测设备的工作状态信息的信息包
    PackageGetInfo:tByteArray;
    iLen:integer;
end;
trPackageSetConfigure=record //监测单元的配置信息
    strPhone:string[cnPhoneLen]; //监测单元的电话号码
    iPhoneLen:integer;           //电话号码的长度
    iUnitsNum:integer;           //监控的列架数目
    PackageSetConfigure:tByteArray;
    iLen:integer;
end;
trPackageUnitsStatus=record //列架工作状态信息包，来自于监测单元
    iPhoneLen:integer;           //电话号码的长度
    strPhone:string[cnPhoneLen]; //监测单元的电话号码
    iUnitsNum:integer;           //监控的列架数目
    PackageUnitsStatus:tByteArray; //列架的工作状态
    strStatus:string[cnMaxNumStatusBytes]; //内容与 PackageUnitsStatus 相同
    vStatus:Variant;           //内容与 PackageUnitsStatus 相同，只是数据类型不同
    iHasBadUnit:integer;        //1 表示无坏单元
    iBadUnitsNum:integer;        //坏单元的数目
    iLen:integer;               //信息包的长度
end;

trPackageSending=record //将要发送的信息包。所有的生成的信息包，先拷贝
//到此信息包中，然后由程序发送该信息包
    PackageGeneral:tByteArray;
    iLen:integer;
end;
trPackageReceived =trPackageSending; //接收到的信息包

trPackageAck =record //确认包，由接收到的信息包生成的确认包。发往通
//信的另一方，表示自己已经接收到正确的信息包

```



```

PackageAck:   tByteArray;
iLen:integer;
end;

```

trPackageIdealAck =record //Acknowledge, 理想的确认信息包, 由将发送的信息包
 //生成的确认信息包。用于确认自己的发送给另一方的包
 //是否正确。在接收到另一方返回的确认包后, 将此
 //理想信息包与之比较, 如果相等表示本次信息包的传送
 //正确无误, 否则, 要考虑重新发送信息包

```

PackageAck:   tByteArray;
iLen:integer;
end;

```

各数据库表单 (Table) 对应的 SQL 语句或者数据库表 (Table) 见表 8.8。
 数据库表单 (Table) 的窗体如图 8.9 所示。

表 8.8 各数据库表单 (Table) 对应的 SQL 语句或者数据库表 (Table)

表名	对应的 SQL 语句或者数据库表 (Table)
ADOQueryDescription	select * from UnitsDescription
ADOTableCenter	CENTER
ADOTableModifyUnits	Monitors
ADOTableShowUnits	Monitors
ADOTableTime	Monitors
ADOTableUnits	Monitors
ADOCommand	select * from monitors
ADODataSetHistoryInfo	select * from UnitsHistory
ADOTWriterAlertInfo	AlertInfo
ADOTWriterCenter	Center
ADOTWriterMonitors	Monitors
ADOTWriterUnitsHistory	UnitsHistory

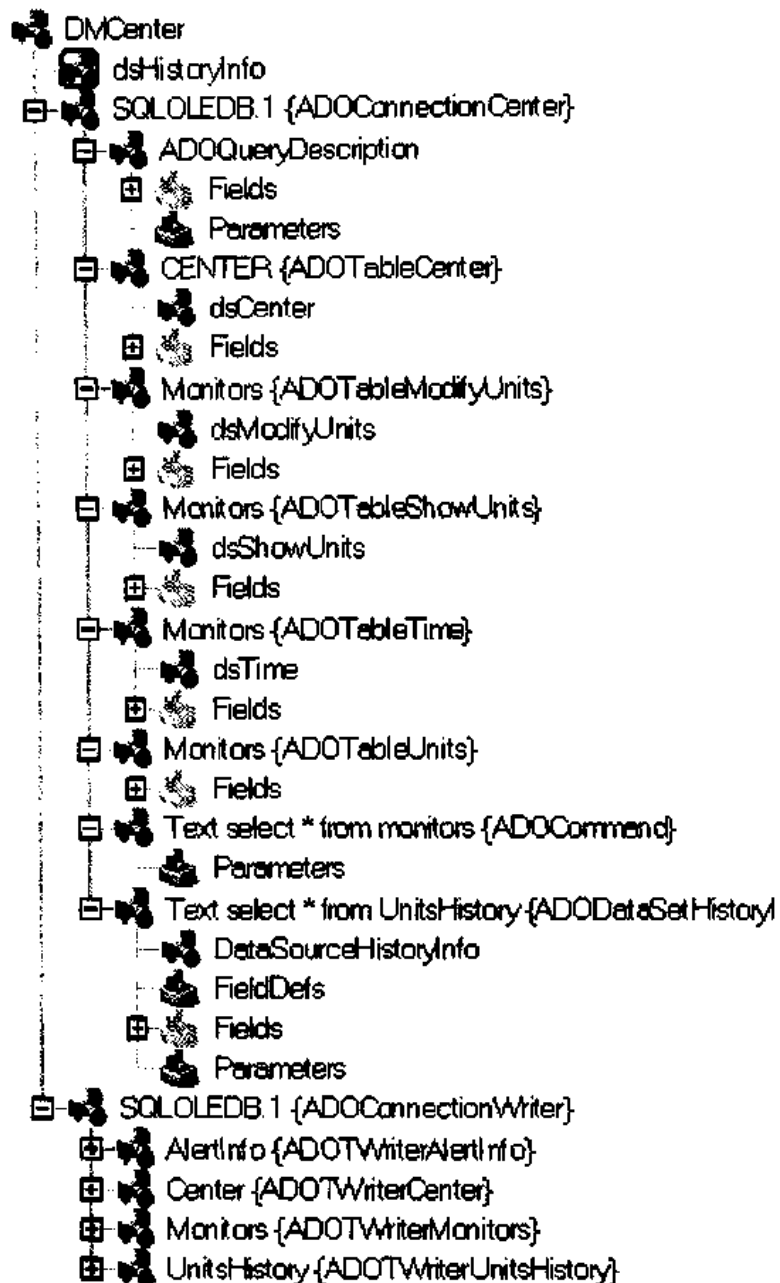


图 8.9 数据库表单 (Table) 的窗体

1. 生成指定通信时间的信息包

//目的: 生成指定通信时间的信息包

//Parameter: aCommTime 指定的通信时间

//结果: 生成指定通信时间的信息包

```
procedure GeneratePackageSetCommTime(const aCommTime:tDateTime);
var Hour, Min, Sec, MSec: Word;
    i:integer;
begin
```

```

DecodeTime(aCommTime, Hour, Min, Sec, MSec);
with aPackageSetCommTime do
begin
    PackageSetCommTime[iArrayBase+1]:=ord('A');    //信息包的首部 (header)
    PackageSetCommTime[iArrayBase+2]:=cnPackageLen; //信息包的长度
    PackageSetCommTime[iArrayBase+3]:=Hour;         //信息包的主体内容
    PackageSetCommTime[iArrayBase+4]:=Min;
    PackageSetCommTime[iArrayBase+5]:=Sec;
    for i:=iArrayBase+6 to cnPackageLen-iCRCNum do //填入空值 (Fill in the blank value)
    begin
        PackageSetCommTime[i]:=cnBlank;
    end;
    iLen:=cnPackageLen-iCRCNum;
    GenerateCRC(PackageSetCommTime,cnPackageLen);
    iLen:=iLen+iCRCNum;    //增加 crc 代码 (code)
end;
end;

```

2. 生成指定可用的电话号码的信息包

//目的: 生成指定可用的电话号码的信息包

//Parameter: rAvailablePhones 可用的电话号码

//结果: 生成可用电话号码的信息包

```

procedure GeneratePackageSetPhones(const rAvailablePhones:trAvailablePhones);
var iPhonesNum:integer;
    iPhoneLen:integer;
    i,j:integer;
    strTemp:string[cnPhoneLen];
begin
    iPhonesNum:=rAvailablePhones.iPhonesNum;
    iPhoneLen:=rAvailablePhones.iPhoneLen;
    with aPackageSetPhones do
    begin
        PackageSetPhones[iArrayBase+1]:=ord('B');    //信息包的首部 (header)
        PackageSetPhones[iArrayBase+2]:=cnPackageLen; //信息包的长度
        PackageSetPhones[iArrayBase+3]:=iPhonesNum;   //电话号码的数目
        PackageSetPhones[iArrayBase+4]:=iPhoneLen;    //电话号码的长度
        for i:=1 to iPhonesNum do    //增加可用的电话号码到信息包
        begin
            strTemp:=rAvailablePhones.aPhones[i];

```

```

    for j:=1 to iPhoneLen do
    begin
        PackageSetPhones[iArrayBase+4+(i-1)*iPhoneLen+j]:=Ord(strTemp[j]);
    end;
end;
for i:=iArrayBase+4+iPhonesNum*iPhoneLen+1 to cnPackageLen-iCRCNum do
begin
    PackageSetPhones[i]:=cnBlank;           //空值
end;
iLen:=cnPackageLen-iCRCNum;
GenerateCRC(PackageSetPhones,cnPackageLen);
iLen:=iLen+iCRCNum;                         //增加 crc 代码
end;// with 语句的结束
end;

```

3. 生成校正时钟信息包

//生成校正时钟信息包

//Parameter: 指定的时钟

//结果: 生成具有校正时钟值的信息包

```
procedure GeneratePackageSetDateTime(const aDateTime:TDateTime);
```

```
var
```

```
    Year, Month, Day, Hour, Min, Sec, MSec: Word;
```

```
    Present: TDateTime;
```

```
    i:integer;
```

```
begin
```

```
    Present:=aDateTime;
```

```
    DecodeDate(Present, Year, Month, Day);
```

```
    DecodeTime(Present, Hour, Min, Sec, MSec);
```

```
    with aPackageSetDateTime do
```

```
    begin
```

```
        PackageSetDateTime[iArrayBase+1]:=Ord('D'); //信息包的首部 (header)
```

```
        PackageSetDateTime[iArrayBase+2]:=cnPackageLen; //信息包的长度
```

```
        PackageSetDateTime[iArrayBase+3]:=Hi(Year); // B:= Hi($1234), 结果为$12
```

```
        PackageSetDateTime[iArrayBase+4]:=Lo(Year);// B := Lo($1234), 结果为$34
```

```
        PackageSetDateTime[iArrayBase+5]:=Month;
```

```
        PackageSetDateTime[iArrayBase+6]:=Day;
```

```
        PackageSetDateTime[iArrayBase+7]:=Hour;
```

```
        PackageSetDateTime[iArrayBase+8]:=Min;
```

```
        PackageSetDateTime[iArrayBase+9]:=Sec;
```

```

    for i:=iArrayBase+10 to cnPackageLen-iCRCNum do
    begin
        PackageSetDateTime[i]:=cnBlank;           //空值
    end;
    iLen:=cnPackageLen-iCRCNum;
    GenerateCRC(PackageSetDateTime,cnPackageLen);
    iLen:=iLen+iCRCNum;                           //增加 crc 代码 Add codes
end;                                              // with 语句的结束
end;

```

4. 生成指定立即发送被测设备的工作状态信息的信息包

//生成指定立即发送被测设备的工作状态信息的信息包

```

procedure GeneratePackageGetInfo;
var i:integer;
begin
    with aPackageGetInfo do
    begin
        PackageGetInfo[iArrayBase+1]:=Ord('C'); //信息包的首部 (header)
        PackageGetInfo[iArrayBase+2]:=cnPackageLen; //信息包的长度
        for i:=iArrayBase+3 to cnPackageLen-iCRCNum do
        begin
            PackageGetInfo[i]:=cnBlank;           //空值
        end;
        iLen:=cnPackageLen-iCRCNum;
        GenerateCRC(PackageGetInfo,cnPackageLen);
        iLen:=iLen+iCRCNum;                       //增加 crc 代码
    end;
end;

```

5. 生成指定监测单元的配置信息包

//生成指定监测单元的配置信息包

//输入参数: strPhone 为监测单元的电话号码

```

procedure GeneratePackageSetConfigure(const strPhone:string);
var strTmp:string;
    i:integer;
begin
    //DMCenter.ADOConnectionWriter.Connected:=false;
    //DMCenter.ADOConnectionWriter.Connected:=true;
    //DMCenter.ADOTWriterMonitors.Active:=True;
    DMCenter.ADOConnectionWriter.Connected:=False;

```

```

if not DMCenter.ADOConnectionWriter.Connected then
    DMCenter.ADOConnectionWriter.Connected:=True;
if not DMCenter.ADOConnectionWriter.Connected then
begin
    exit;
end;

//激活数据库
if not DMCenter.ADOTWriterMonitors.Active then
    DMCenter.ADOTWriterMonitors.Active:=True;
//如果数据库存在的话
if DMCenter.ADOTWriterMonitors.Active then
begin
    DMCenter.ADOTWriterMonitors.Filtered:=False;
    DMCenter.ADOTWriterMonitors.Refresh;

    //首先找到该电话所在的记录
if DMCenter.ADOTWriterMonitors.Locate('PhoneNum',strPhone,[loCaseInsensitive]) then
begin
    //从该记录中提取配置信息
    aPackageSetConfigure.strPhone:=strPhone;
    aPackageSetConfigure.iPhoneLen:=Length(strPhone);
    aPackageSetConfigure.iUnitsNum:=
DMCenter.ADOTWriterMonitors.FieldByName('UnitsNum').AsInteger;
    //vTmp:=varArrayCreate([1,cnMaxNumStatusBytes],varByte);
strTmp:=DMCenter.ADOTWriterMonitors.FieldByName
('UnitsConfiguration').AsString;
    //将配置信息，电话号码写入数据包
    aPackageSetConfigure.PackageSetConfigure[1]:=Ord('E');
    //整个信息包的长度
    aPackageSetConfigure.PackageSetConfigure[2]:=cnPackageLen;
    //电话号码的长度
    aPackageSetConfigure.PackageSetConfigure[3]:=aPackageSetConfigure.iPhoneLen;
    //配置信息的长度=监控的列架的数目
    aPackageSetConfigure.PackageSetConfigure[4]:=
aPackageSetConfigure.iUnitsNum div 256;
    aPackageSetConfigure.PackageSetConfigure[5]:=
aPackageSetConfigure.iUnitsNum mod 256;
    //电话号码

```

```

    for i:=1 to aPackageSetConfigure.iPhoneLen do
    begin
        aPackageSetConfigure.PackageSetConfigure[5+i]:=
ord(aPackageSetConfigure.strPhone[i]);
    end;
    //配置信息
    if Length(strTmp)>=cnMaxNumStatusBytes then
    begin
        for i:=1 to cnMaxNumStatusBytes do
        begin
            aPackageSetConfigure.PackageSetConfigure
[5+aPackageSetConfigure.iPhoneLen+i]:= Ord(strTmp[i]);
        end;
    end
    else //Length(strTmp)<cnMaxNumStatusBytes
    begin
        for i:=1 to cnMaxNumStatusBytes do
        begin
            aPackageSetConfigure.PackageSetConfigure
[5+aPackageSetConfigure.iPhoneLen+i]:=0;
            // Ord(strTmp[i])
        end
    end;
    aPackageSetConfigure.iLen:=cnPackageLen-iCRCNum;
    GenerateCRC(aPackageSetConfigure.PackageSetConfigure,cnPackageLen);
    aPackageSetConfigure.iLen:=aPackageSetConfigure.iLen+iCRCNum;//增加 crc 代码
end;
end;
end;

```

6. 解码配置信息包

//如果信息包能被正确解码，则返回 True，否则返回 False
 //输入参数：aLocalPackageReceived 接收到的信息包
 //结果：由接收到的信息报解码成配置信息，并将信息存入数据库中

```

function DecodeSetConfigurePackage(const
aLocalPackageReceived:trPackageReceived):Boolean;//配置信息记录
var strPhone,strTmp:string;
    i,iPhoneLen,iUnitsNum:integer;
    vTmp:Variant;

```

```

begin
    Result:=True;
    DMCenter.ADOConnectionWriter.Connected:=False;
    if not DMCenter.ADOConnectionWriter.Connected then
        DMCenter.ADOConnectionWriter.Connected:=True;
    if not DMCenter.ADOConnectionWriter.Connected then
        begin
            Result:=False;
            exit;
        end;

    if aLocalPackageReceived.PackageGeneral[1]<>ord('E') then
//如果信息包的命令码为'E'，表示此信息包为正确的信息包
    begin
        Result:=False;
        exit;
    end;
    //获取电话号码长度
    iPhoneLen:=aLocalPackageReceived.PackageGeneral[3];
    //获取配置信息的长度
    iUnitsNum:=aLocalPackageReceived.PackageGeneral[4];
    iUnitsNum:=iUnitsNum*256+aLocalPackageReceived.PackageGeneral[5];
    //获取电话号码
    strPhone:="";
    for i:=1 to iPhoneLen do
        begin
            strPhone:=strPhone+Chr(aLocalPackageReceived.PackageGeneral[5+i]);
        end;
    //获取配置信息
    strTmp:="";
    vTmp:=VarArrayCreate([1,cnMaxNumStatusBytes],varByte);
    for i:=1 to cnMaxNumStatusBytes do
        begin
            strTmp:=strTmp+Chr(aLocalPackageReceived.PackageGeneral[5+iPhoneLen+i]);
            vTmp[i]:=Ord(Chr(aLocalPackageReceived.PackageGeneral[5+iPhoneLen+i]));
        end;
    //将数据存入数据库，激活数据库
    if not DMCenter.ADOTWriterMonitors.Active then
        DMCenter.ADOTWriterMonitors.Active:=True;

```



```

//如果数据库存在的话
if DMCenter.ADOTWriterMonitors.Active then
begin
    DMCenter.ADOTWriterMonitors.Filtered:=False;
    DMCenter.ADOTWriterMonitors.Refresh;
    //首先找到该电话所在的记录
    if DMCenter.ADOTWriterMonitors.RecordCount>0 then
        if DMCenter.ADOTWriterMonitors.Locate
('PhoneNum',strPhone,[loCaseInsensitive]) then
            begin
                DMCenter.ADOTWriterMonitors.Edit;
                //向该记录中写入配置信息
                DMCenter.ADOTWriterMonitors.FieldByName
('UnitsConfiguration').AsVariant:=vTmp;
                //写入列架数
                DMCenter.ADOTWriterMonitors.FieldByName
('UnitsNum').AsInteger:=iUnitsNum;
                DMCenter.ADOTWriterMonitors.Post;
                DMCenter.ADOTWriterMonitors.Refresh;
            end
            else Result:=False;
        end
    else Result:=False;

end;

```

7. 解码状态信息包

//如果信息包能被正确解码，则返回 True，否则返回 False

//输入参数：aLocalPackageReceived，接收到的信息包

//结果：由接收到的信息报解码成状态信息记录

```
function DecodeStatusInfoPackage(const aLocalPackageReceived:trPackageReceived
                                ):Boolean;//状态信息记录
```

// aLocalPackageReceived 为接收的信息包

```
var  inti:integer;
```

```
    iBadUnitsNum:integer;
```

```
    iTmp:integer;
```

```
    byteTmp:byte;
```

//输入参数：chStatus:含有状态信息的字节

// ishrNum: 右移的位数

```

//对于二进制数 101099, 如果 ishrNum 为 5, 则输出结果为 1
//结果: 计算 chStatus 中, 从右往左计算其中含有的 0 的个数
function CalcBadUnitsNum(const chStatus:char;ishrNum:integer):integer;
var i,iTemp:integer;
    iLocalBadUnitsNum:integer;
begin
    iTemp:=ord(chStatus);
    iLocalBadUnitsNum:=0;
    for i:=1 to ishrNum do
        begin
            if (iTemp and $01)<>$01 then Inc(iLocalBadUnitsNum);
//1 表示工作正常, 0 表示工作异常
            iTemp:=iTemp shr 1;
        end;
        Result:=iLocalBadUnitsNum;
    end;

begin
{   trPackageUnitsStatus:=record //列架工作状态信息包, 来自于监测单元
    iPhoneLen:integer;    //电话号码的长度
    strPhone:string[cnPhoneLen];//监测单元的电话号码
    iUnitsNum:integer;    //监控的列架数目
    PackageUnitsStatus:tByteArray;//列架的工作状态
    strStatus:string;//内容与 PackageUnitsStatus 相同, 只是数据类型不同
    iHasBadUnit:integer; //1 表示无坏单元
    iBadUnitsNum:integer;//坏单元的数目
    iLen:integer;        //信息包的长度
end;
tByteArray=array [1..cnPackageLen] of byte;}
if aLocalPackageReceived.PackageGeneral[1]<>ord('F') then
//如果信息包的命令码为'F', 表示此信息包为正确的信息包
begin
    Result:=False;
    exit;
end;
aPackageUnitsStatus.iLen:=aLocalPackageReceived.PackageGeneral[2];
//信息包的长度
aPackageUnitsStatus.iPhoneLen:=aLocalPackageReceived.PackageGeneral[3];
//电话号码的长度

```

```

iTmp:=aLocalPackageReceived.PackageGeneral[4];      //Hi
iTmp:=iTmp*256;
iTmp:=iTmp+aLocalPackageReceived.PackageGeneral[5];  //Lo
aPackageUnitsStatus.iUnitsNum:=iTmp;                //监控的列架数目

for inti:=1 to aPackageUnitsStatus.iPhoneLen do      //监测单元的电话号码
begin
aPackageUnitsStatus.strPhone[inti]:=
chr(aLocalPackageReceived.PackageGeneral[5+inti]);
//使用字符串
end;
//设置电话号码长度到字符串中, 注意字符串的第 0 个位置放置字符串长度值
aPackageUnitsStatus.strPhone[0]:=Chr(aPackageUnitsStatus.iPhoneLen);
//SetLength(aPackageUnitsStatus.strPhone,aPackageUnitsStatus.iPhoneLen)

iBadUnitsNum:=0;
aPackageUnitsStatus.vStatus:=VarArrayCreate([1,cnMaxNumStatusBytes], VarByte);
aPackageUnitsStatus.strStatus:="";
for inti:=1 to cnMaxNumStatusBytes do                //列架的工作状态
begin
byteTmp:=
aLocalPackageReceived.PackageGeneral[5+aPackageUnitsStatus.iPhoneLen+inti];
aPackageUnitsStatus.PackageUnitsStatus[inti]:=byteTmp;
aPackageUnitsStatus.vStatus[inti]:=Ord(Chr(byteTmp));
aPackageUnitsStatus.strStatus:=aPackageUnitsStatus.strStatus+Chr(byteTmp);
end;

//计算列架的工作状态所指示的工作异常的列架数目
for inti:=1 to (aPackageUnitsStatus.iUnitsNum div 8) do //列架的工作状态
iBadUnitsNum:=
iBadUnitsNum+CalcBadUnitsNum(aPackageUnitsStatus.strStatus[inti],8);
if (aPackageUnitsStatus.iUnitsNum mod 8)>0 then
iBadUnitsNum:=iBadUnitsNum+CalcBadUnitsNum
(Chr(aPackageUnitsStatus.PackageUnitsStatus[inti]),
(aPackageUnitsStatus.iUnitsNum mod 8));

// SetLength(aPackageUnitsStatus.strStatus,cnMaxNumStatusBytes)
aPackageUnitsStatus.iBadUnitsNum:=iBadUnitsNum;
if iBadUnitsNum>0 then

```

监控中心的主流程如图 8.10 所示。监控中心启用两个线程查询 Modem。一个查询主用 Modem，另一个查询备用 Modem。当需要发送命令到指定的监测单元时，如果发现当前备用 Modem 正在使用，则强行中止通信，再用备用 Modem 发送命令。

信息包发送的交互图如图 8.11 所示。首先监控中心发送的控制指令传到监控中心的

-215-

Modem, 再传到监测单元的 Modem, 监测单元接收该控制指令。监测单元校验接收的指令, 如果指令包的 CRC 码正确, 则监测单元发送确认信息给监控中心, 监控中心接收到正确的确认信息, 将 Modem 挂机, 指令发送完毕。如果监测单元发现接收的指令有误, 则抛弃该指令, 同时将 Modem 挂机; 监控中心等待指定的时间, 未接收到反馈信息, 则将 Modem 挂机, 准备重发控制指令。

监控中心到监测单元的Interaction图

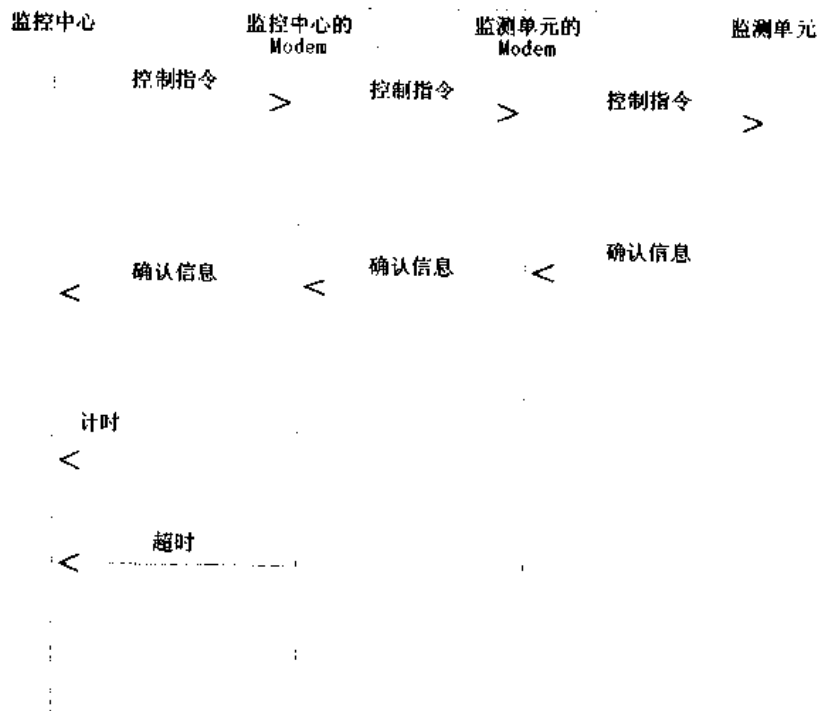


图 8.11 信息包发送的交互图

发送模块的总流程如下:

生成控制信息包→建立连接→发送数据

(1) 生成控制信息包

控制信息包包括如下部分。

□ 指定通信时间: 命令代码为 A, 其中通信时间格式为 HH:MM:SS, 要求用正确的值填入相应的字段, 例如用值 12:30:40 转换的 16 进制 0C:1E:28 填入。

□ 指定可用的电话号码: 命令代码为 B。用于通知监测单元监控中心的可用电话号码。可用电话号码按主用电话号码、备用电话号码的顺序填写在该信息包中。监测单元在与监控中心通信时, 首先使用主要电话号码与监控中心通信, 如果发现监控中心的主要电话忙, 则试图使用备用电话号码与监控中心通信。

□ 指定立即发送被测设备的工作状态信息: 命令代码为 C。用户通过该命令可以要求指定的监测单元立即发送被测设备的工作状态信息。由于 Modem 主要用于接收监测单元的信息包, 所以这个命令通过备用 Modem 发送给指定的监测单元。

□ 校正时钟：命令代码为 D。考虑到监测单元的时钟可能与监控中心的时钟不一致，因而不能在监控中心指定的时间发送被测设备的工作状态信息，所以使用监控中心的时钟校正监测单元的时钟。

□ 设定监测单元的配置信息：命令代码为 E。用户可以通过监控中心配置监测单元的配置信息。

生成控制信息包的图如图 8.12 所示。

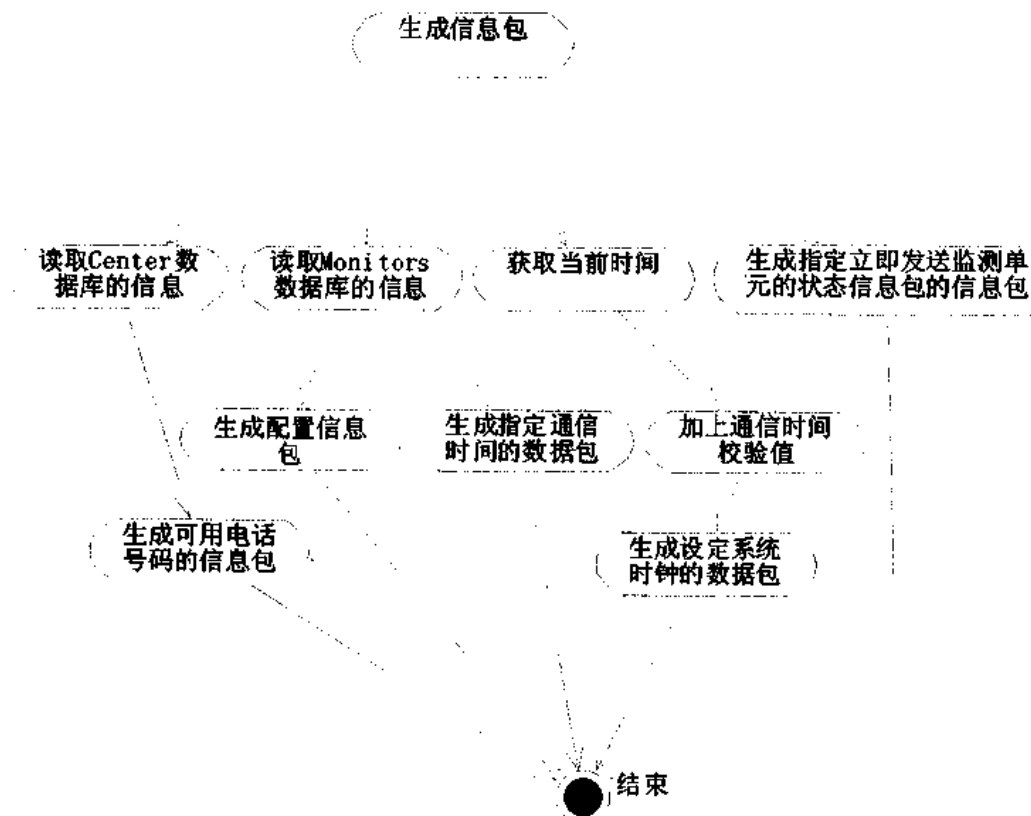


图 8.12 生成控制信息包

(2) 建立连接

建立连接的流程图如图 8.13 所示。

(3) 发送数据

发送数据的流程图如图 8.14 所示。

(4) 发送数据

发送数据的流程图如图 8.14 所示。

其中的处理人工测试的流程图如图 8.15 所示。

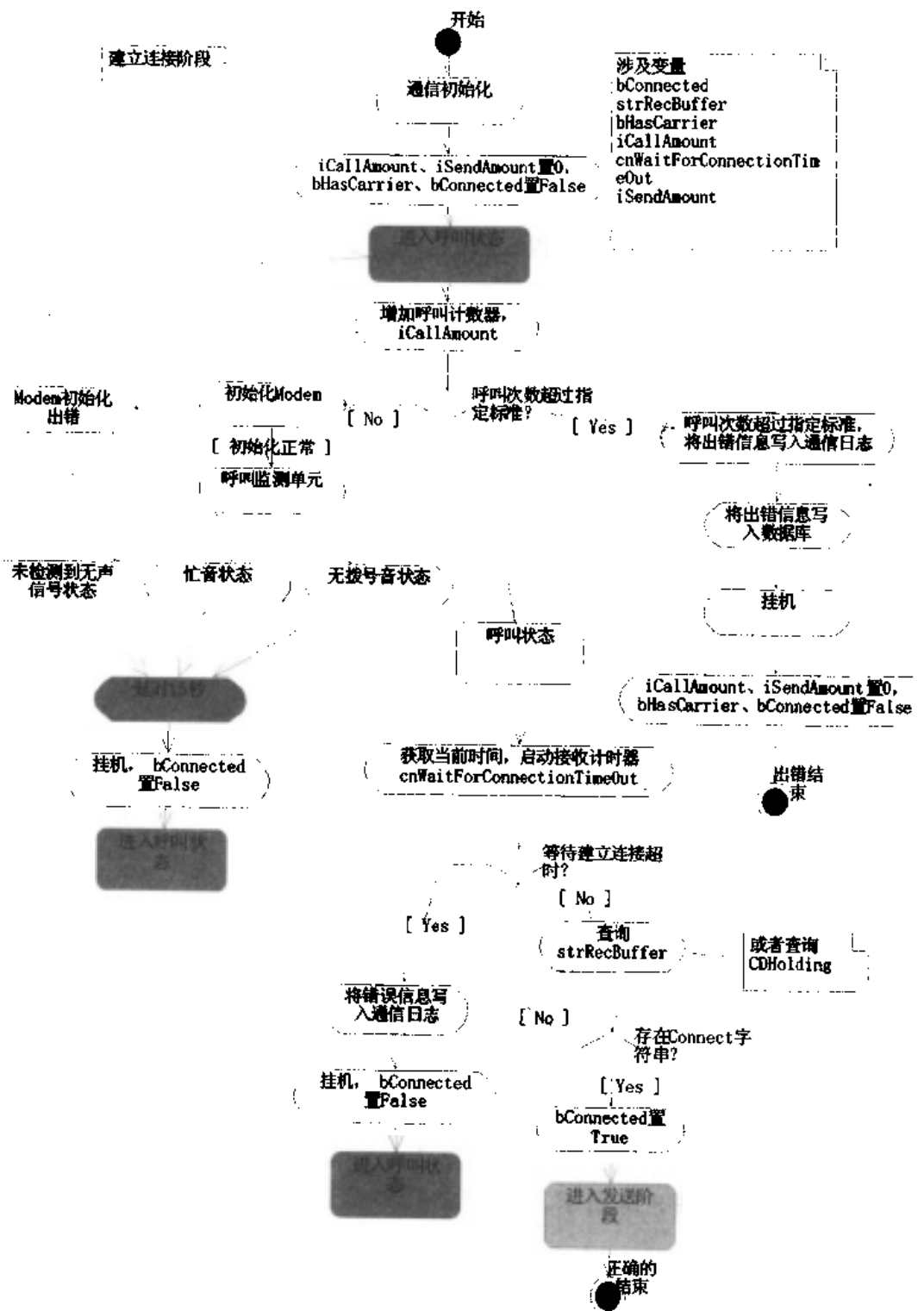


图 8.13 建立连接的流程图

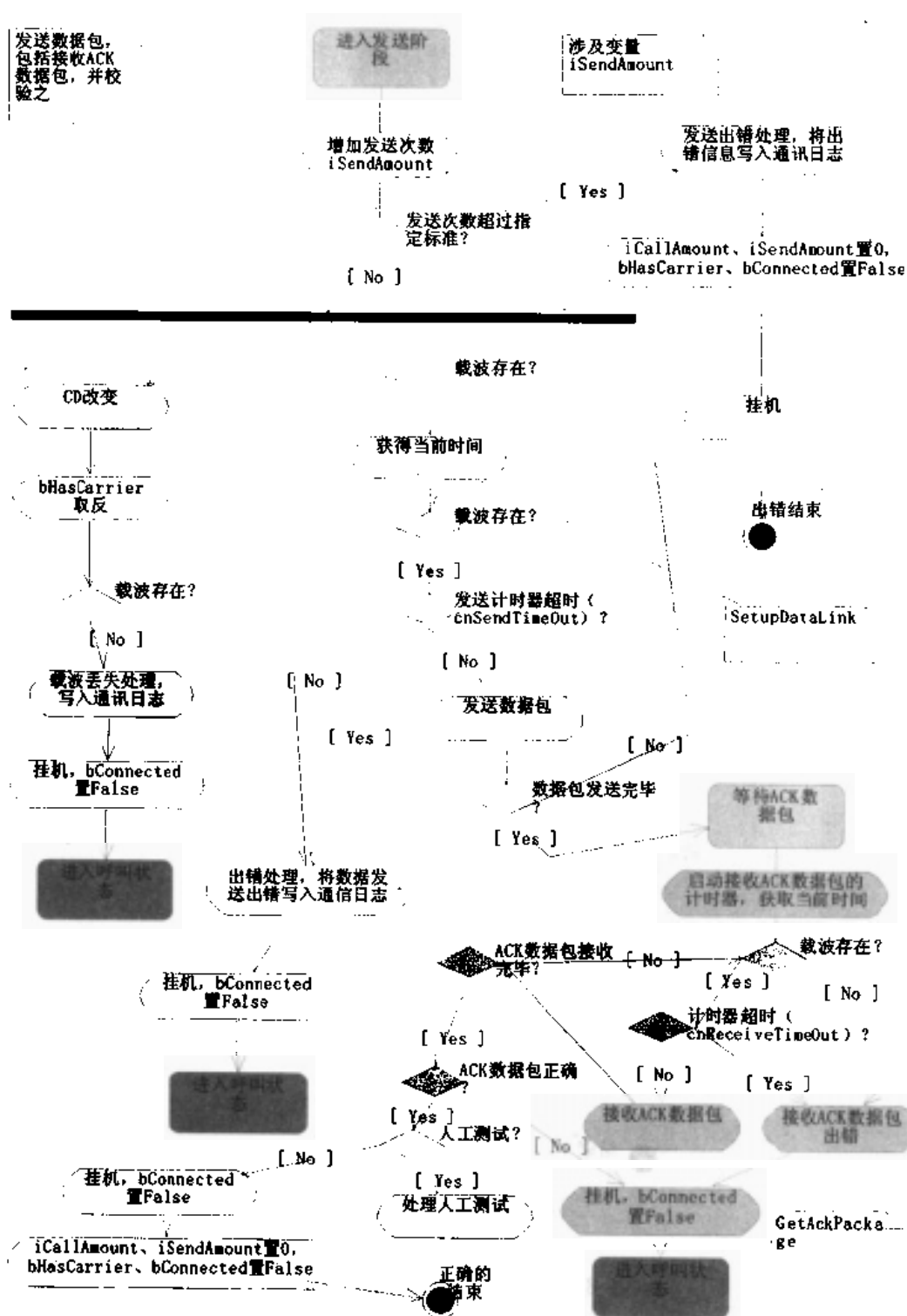


图 8.14 发送数据的流程图

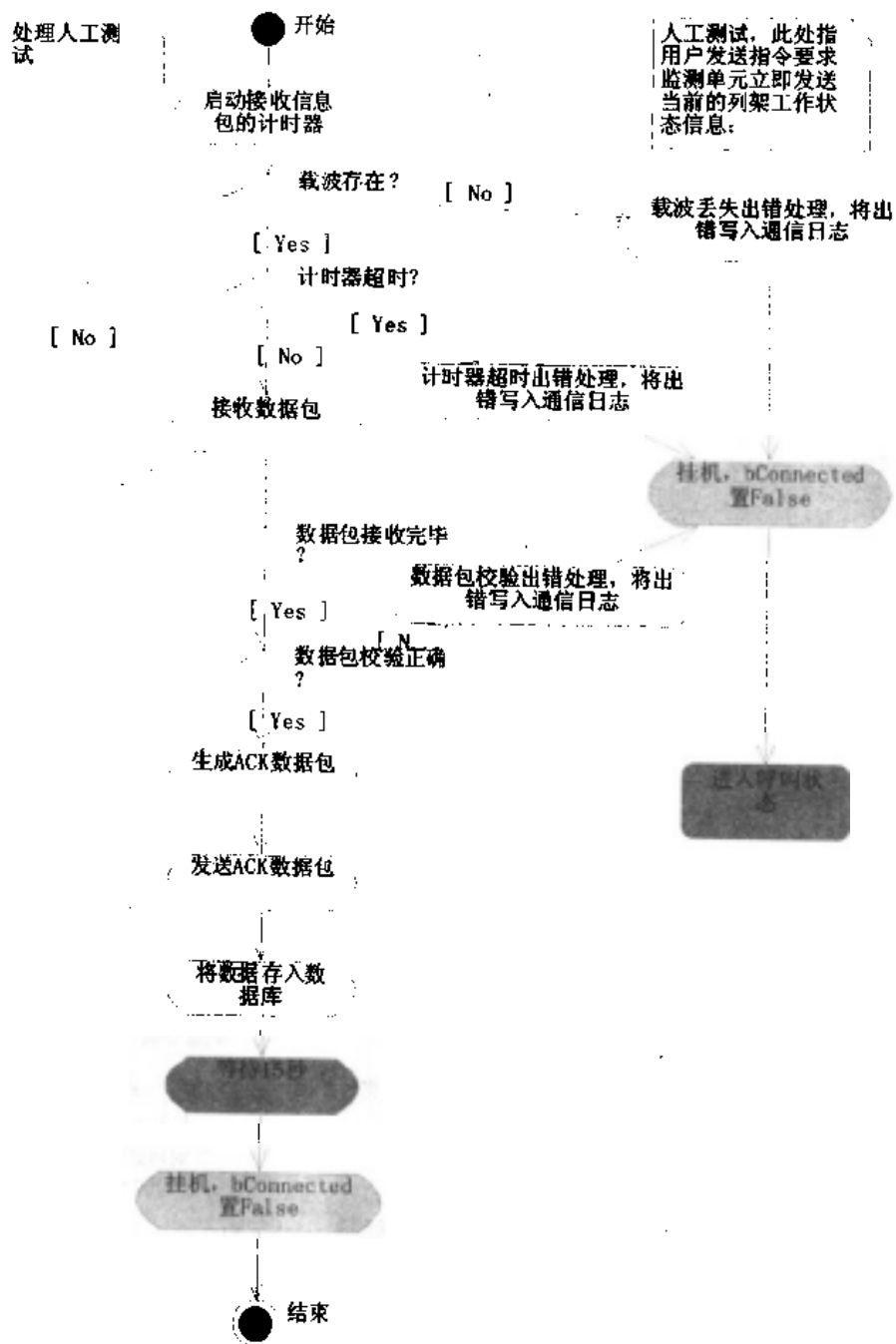


图 8.15 处理人工测试的流程图

单元 `uSendThread` 的类关系图如图 8.16 所示。



图 8.16 单元 uSendThread 的类关系图

发送线程使用 MSComm 控件控制 Modem，与其中监控中心的一个接收线程互斥地使用同一个 Modem。监控中心使用至少两个 MSComm 控制分别控制 Modem，其中 MSComm 控件的设置如图 8.17 所示。一个 MSComm 控件对应一个串口。

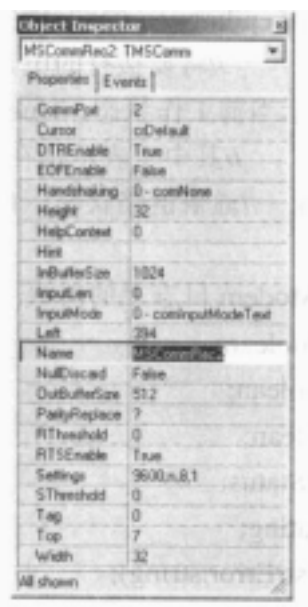


图 8.17 MSComm 控件的属性设置

下面为具体代码:

```
unit uSendThread;
//发送线程类
//用于发送指定的信息包
//关于发送流程请参见 ROSE 2000 下的集中监控系统设计文档
interface
uses
  Classes,OleCtrls,MSCommLib_TLB, ComCtrls,Windows,Sysutils,
  cModem,Packages,DM,db;
const cnCallAmountMax=3;           //对一个电话号码允许的最大呼叫次数
const cnSendAmountMax=3;           //一个数据包最多允许的发送次数
const cnGetAckTimeOut=10000;        //接收 ACK 数据包的最大时间
const cnGetStatusInfoTimeOut=4000; //接收状态信息包的最大时间
const cnSetupDataLinkTimeOut=60000; //建立数据链路的最大时间
const cnWaitBeforeHangUp=1500;     //在挂机之前的等待时间
type
  TSendThread = class(TThread)
  private
    { Private declarations }
    faModem:TModem;
    fiSendAmount:integer;           //发送次数
    fiCallAmount:integer;           //呼叫次数
    fbHasCarrier:Boolean;           //True 表示有载波
    fbConnected:Boolean;            //True 表示已经与远端的 Modem 连接
    faPackageSending:trPackageSending; //欲发送的数据包
    fstrPhone:string;               //将要拨打的电话号码
    fbIsAutoTest:Boolean; //表示当前工作状态, True 表示自动测试, False 表示人工测试
    fbCommunicating:Boolean;        //其中 1 表示正在与该监测单元通信
    fbCommLinkStatus:boolean; //通信链路状况。1 表示工作正常, 0 表示工作异常
//即不能通信
    fstrPrefixion:string; //加于 Modem 日志上的前缀
    function GetAckPackage:Boolean;
    function SetupDataLink:Boolean;
    function GetStatusInfo:Boolean;
    procedure MarkCommLinkStatus;
    procedure MarkCommunicating;
    procedure DealError(const strError:string);
  protected
    procedure Execute; override;
```

```

    destructor Destroy;override;
public
    fMsComm:TMSComm;
    bSendSuccessfully:Boolean;//True 表示发送成功
constructor Create(aMsCOMM:TMSCOMM;const aPackageSending:trPackageSending;
                    const strPhone:string;const bIsAutoTest:Boolean);
end;

//系统中只能有一个发送线程，所以在生成一个新的发送线程之前，
//请判断是否已经存在一个发送线程，
//如果要强行生成一个发送线程，您可以终止当前的发送线程，然后
//在生成一个发送线程
var baSendThreadIsRunning:Boolean;
    iCRCBadNums:integer;//CRC 出错字数
    iTotalCommNums:integer;//总的通信次数
implementation

destructor TSendThread.Destroy;
begin
    faModem.HangUpModem;//挂机
    //标明系统与指定监测单元的通信结束
    fbCommunicating:=False;
    MarkCommunicating;//1 表示正在与该监测单元通信
    //标明一个发送线程工作结束
    baSendThreadIsRunning:=False;
    // inherited 释放
end;

//建立数据链路
//返回值: True 成功建立数据链路, False 建立数据链路失败
function TSendThread.SetupDataLink:Boolean;
var dwSetupStart,dwSetupEnd:DWord;
//建立数据链路的开始时间, 建立数据链路的结束时间
    iReturnValue:integer;
    iTmp:integer;
begin
    faModem.ShowModemInfo(fstrPrefixion+'开始与监测单元'+fstrPhone+'建立数据链路!');
    Result:=True;
    while (not Terminated) and (not fbConnected) do//大循环

```

```

begin
  Inc(fiCallAmount);
  if fiCallAmount>cnCallAmountMax then
  begin
    Result:=False;
    DealError('呼叫次数超过指定的标准，终止本次通信！');
    exit;
  end;
  if not faModem.InitCommunication then Continue;//开始下一个循环
    // Continue 影响控制流程以进行下一个反复
  faModem.ShowModemInfo(fstrPrefixion+'第'+IntToStr(fiCallAmount)+'次呼叫！');
  faModem.DialAModem(fstrPhone);
  //获取建立数据链路的开始时间
  dwSetupStart:=GetTickCount;
  while (not Terminated) and (true) and (fMsComm.PortOpen) and
(fMsComm.DSRHolding) do//小循环
  begin
    //等待 Modem 响应，时间不应太短，否则程序不能正确判断来自 Modem 响应
    //Sleep(1000)，检测要等待的时间，否则程序能拨号了
    iTmp:=0;
    while (not Terminated) and (iTmp<60) do
    begin
      sleep(20);
      Inc(iTmp,1);
    end;
    //获取建立数据链路的结束时间
    dwSetupEnd:=GetTickCount;
    //分析 Modem 的响应
    iReturnValue:=faModem.AnalyzeModemResponseCode;
    if
      iReturnValue=MODEM_NODIALTONE then
    begin
      faModem.ShowModemInfo(fstrPrefixion+'无拨号音！');
      break;//跳出小循环，开始下一个循环
    end;
    if //iReturnValue=MODEM_NOCARRIER 或者
      iReturnValue=MODEM_BUSY then
    begin
      faModem.ShowModemInfo(fstrPrefixion+'线路忙！');

```

```

        break;//跳出小循环, 开始下一个循环
    end;

    //判断建立数据链路是否超时
    if dwSetupEnd-dwSetupStart>cnSetupDataLinkTimeOut then
    begin
        Result:=False;
        faModem.ShowModemInfo(fstrPrefixion+'等待对方响应超时! ');
        break;//跳出小循环, 开始下一个循环
    end;
    //侦测到载波
    if fMsComm.CDHolding then break;//跳出小循环, 开始下一个循环
end;

if fMsComm.CDHolding then
begin
    faModem.ShowModemInfo(fstrPrefixion+'数据链路建立! ');
    fbConnected:=True;
    Result:=True;
end
else//Modem 无法响应
begin
    Sleep(cnWaitBeforeHangUp);
    faModem.HangUpModem;
    fbConnected:=False;
end;
end;//while not fbConnected do 语句结束
end;

//线程初始化
//输入参数: aPackageSending 将要发送的数据包
//          strPhone: 将要通信的目标监测单元的电话号码
//          blsAutoTest: 表示当前的工作状态, True: 自动测试, False: 人工测试
//所谓人工测试, 即生成一个要求监测单元立即发送状态包的信息包, 用发送线程
//发送给监测单元
constructor TSendThread.Create(aMsCOMM:TMSCOMM;
const aPackageSending:trPackageSending;
const strPhone:string;const blsAutoTest:Boolean);
begin

```

```

//初始化各变量
//标明一个发送线程正在工作
    baSendThreadIsRunning:=True;
    fbIsAutoTest:=bIsAutoTest;
    fMsComm:=aMsComm;
    fstrPrefixion:='串口'+IntToStr(fMsComm._CommPort)+'发送线程: ';
    fiSendAmount:=0;
    bSendSuccessfully:=False;
    fstrPhone:=strPhone;
    faModem:=TModem.Create(aMsComm);
    faPackageSending:=aPackageSending;
    inherited Create(False);
end;

//接收确认信息包
// 成功接收, 则返回 True
//      否则返回 False
function TSendThread.GetAckPackage:Boolean;
var dwGetAckStart,dwGetAckEnd:DWord;
//接收数据包的开始时间, 接收数据包的结束时间
    strReceivedSlice:string;
begin
    faModem.ShowModemInfo(fstrPrefixion+'开始接收 ACK 数据包! ');
    Result:=False;
    //获取接收数据包的开始时间
    dwGetAckStart:=GetTickCount;
    while (not Terminated) and True do
    begin
        //载波存在吗
        if not fMsComm.CDHolding then
        begin
            Result:=False;
            faModem.ShowModemInfo(fstrPrefixion+'载波丢失! ');
            break;//跳出循环
        end;

        //获取接收数据包的结束时间
        dwGetAckEnd:=GetTickCount;
        //判断接收 ACK 数据包是否超时

```

```

if (dwGetAckEnd-dwGetAckStart)>cnGetAckTimeOut then
begin//接收超时
    Result:=False;
    faModem.ShowModemInfo(fstrPrefixion+'接收 ACK 数据包超时! ');
    break;//跳出循环
end;

//检查接收到的 ACK 数据包
faModem.GetInfoFromModem(strReceivedSlice,'Binary');
faModem.StoreIntostrRecBuffer(strReceivedSlice);
if faModem.aPackageReceived.iLen>=cnPackageLen then
begin//接收到完整信息包
    faModem.ShowModemInfo(fstrPrefixion+'接收到的信息包: ');
    faModem.ShowModemPackage(faModem.aPackageReceived);

    if CompareTwoArrays(faPackageSending.PackageGeneral,
                        faModem.aPackageReceived.PackageGeneral,
                        faModem.aPackageReceived.iLen) then
    begin
//校验接收到的 ACK 信息包，接收到的 ACK 信息包正确
        Result:=True;
        faModem.ShowModemInfo(fstrPrefixion+'接收到的 ACK 信息包正确! ');
        break;//跳出循环
    end
    else
    begin
//接收到的 ACK 信息包有错
        Result:=False;
        faModem.ShowModemInfo(fstrPrefixion+'接收到的 ACK 数据包有错! ');
        break;//跳出循环
    end;
end;
end;
end;

//输入参数 bCommLinkStatus，用于表示
//通信链路的状况。1 工作正常（缺省值），0 工作异常，即不能通信
//结果：在数据库相应的监测单元记录上标明当前通信链路的状况
procedure TSendThread.MarkCommLinkStatus;

```



```

begin
    DMCenter.ADOConnectionWriter.Connected:=False;
    if not DMCenter.ADOConnectionWriter.Connected then
        DMCenter.ADOConnectionWriter.Connected:=True;
    if not DMCenter.ADOConnectionWriter.Connected then
        begin
            exit;
        end;
    DMCenter.ADOTWriterMonitors.Active:=False;
    DMCenter.ADOTWriterMonitors.Active:=True;
    if DMCenter.ADOTWriterMonitors.Active then
        begin
            if DMCenter.ADOTWriterMonitors.Locate('PhoneNum',fstrPhone,[loPartialKey]) then
                begin
                    DMCenter.ADOTWriterMonitors.Edit;
                    DMCenter.ADOTWriterMonitors.FieldByName('CommLinkStatus').AsBoolean:=
fbCommLinkStatus;
                    //通信链路的状态。1 工作正常（缺省值），0 工作异常，即不能通信
                    DMCenter.ADOTWriterMonitors.Post;
                    DMCenter.ADOTWriterMonitors.Active:=False;
                    DMCenter.ADOTWriterMonitors.Active:=True;
                end;
            DMCenter.ADOTWriterMonitors.Filtered:=False;
        end;
    end;
end;

```

//输入参数：bCommunicating 其中 1 表示正在与该监测单元通信
//结果：在数据库相应的监测单元记录上标明是否正在与该监测单元通信

```

procedure TSendThread.MarkCommunicating;
begin
    DMCenter.ADOConnectionWriter.Connected:=False;
    if not DMCenter.ADOConnectionWriter.Connected then
        DMCenter.ADOConnectionWriter.Connected:=True;
    if not DMCenter.ADOConnectionWriter.Connected then
        begin
            exit;
        end;

    DMCenter.ADOTWriterMonitors.Active:=True;

```

```

if DMCenter.ADOTWriterMonitors.Active then
begin
  if DMCenter.ADOTWriterMonitors.Locate('PhoneNum',
fstrPhone,[loCaseInsensitive]) then
    begin
      DMCenter.ADOTWriterMonitors.Edit;
      DMCenter.ADOTWriterMonitors.FieldByName('Communicating').AsBoolean:=
fbCommunicating;
      //1 表示正在与该监测单元通信
      DMCenter.ADOTWriterMonitors.FieldByName('CurrentWorkStatus').AsBoolean:=True;
      //自动测试
      DMCenter.ADOTWriterMonitors.Post;
      DMCenter.ADOTWriterMonitors.Active:=False;
      DMCenter.ADOTWriterMonitors.Active:=True;
    end;
    DMCenter.ADOTWriterMonitors.Filtered:=False;
  end;
end;

//输入参数: strError 出错信息
//结果: 将出错信息写入通信日志, 挂机, 清空接收缓冲区
//设置各变量值
procedure TSendThread.DealError(const strError:string);
begin
  if Length(strError)>0 then
    faModem.ShowModemInfo(fstrPrefixion+strError);
    faModem.HangUpModem;
    faModem.ClearRecBuffer;
    bSendSuccessfully:=False;
    fbHasCarrier:=False;
    fbConnected:=False;
  end;

  //线程主体
  //整个通信过程均记载到通信日志中
  procedure TSendThread.Execute;
  begin
    //放置线程代码
    faModem.ShowModemInfo(fstrPrefixion+'发送线程初始化成功! ');
  end;
end;

```

```

faModem.ShowModemInfo(fstrPrefixion+'开始发送信息包: ');
faModem.ShowModemPackage(faPackageSending);
//标明系统正在与指定监测单元通信
fbCommunicating:=True;//1 表示正在与该监测单元通信
synchronize(MarkCommunicating);
While (not Terminated) and (not bSendSuccessfully) do
begin
    if not SetupDataLink then
    begin//通信链路的状况。1 工作正常（缺省值），0 工作异常，即不能通信
        fbCommLinkStatus:=False;
        Synchronize(MarkCommLinkStatus);
        break;//跳出循环
    end
    else
    begin    //通信链路建立失败
        fbCommLinkStatus:=True;
        Synchronize(MarkCommLinkStatus);
    end;

    if Terminated then
break;
        Inc(fiSendAmount);
        faModem.ShowModemInfo(fstrPrefixion+'第'+IntToStr(fiCallAmount)+'次发送! ');
        bSendSuccessfully:=False;

        if (not Terminated) and (fiSendAmount<=cnSendAmountMax) then
        begin
            //const aPackageSending:trPackageSending
            bSendSuccessfully:=faModem.SendPackageIntoModem(aPackageSending);
            //处理发送出错
            if not bSendSuccessfully then
            begin
                DealError('发送数据出错! ');
                continue;        //进入呼叫状态
            end;

            //接收 ACK 数据包
            if GetAckPackage then
            begin
                //正确接收到 ACK 数据包

```

```

faModem.ShowModemInfo(fstrPrefixion+'接收到正确的 ACK 数据包! ');
//清空接收缓冲区, 以接收状态信息包或其他信息
faModem.ClearRecBuffer;
//清空接收的数据包
faModem.aPackageReceived.iLen:=0;
//处理人工测试
if not fbIsAutoTest then
    if not GetStatusInfo then
continue;//进入呼叫状态
        DealError('');
        fiCallAmount:=0;
        fiSendAmount:=0;
        break;//跳出循环
    end
else //未能正确接收到 ACK 数据包
begin
    DealError('未能接收到正确的 ACK 数据包! ');
    continue; //进入呼叫状态
end;
end
else //超过指定的发送次数
begin
    DealError('对当前数据包的发送次数超过指定的标准, 停止发送! ');
    fiCallAmount:=0;
    fiSendAmount:=0;
    break; //跳出循环, 退出本线程
end;
end;//While not bSendSuccessfully do
//标明系统与指定监测单元的通信结束
//1 表示正在与该监测单元通信
fbCommunicating:=False;
Synchronize(MarkCommunicating);
//1 表示正在与该监测单元通信, 标明一个发送线程工作结束
baSendThreadIsRunning:=False;
faModem.ShowModemInfo(fstrPrefixion+'终止发送线程! ');
end;

//接收状态信息包, 如果信息包正确, 则将接收的状态信息
//存入到数据库中, 整个接收过程均记载到通信日志中

```

```

//如果能够正确地接收到状态信息包，则返回 True;否则为 False
function TSendThread.GetStatusInfo;
var dwGetStatusInfoStart,dwGetStatusInfoEnd:DWord;
//接收信息包的开始时间，接收信息包的结束时间
    strReceivedSlice:string;
begin
    Result:=False;
    faModem.ShowModemInfo(fstrPrefixion+'正在接收监测单元'+fstrPhone+'的状态信息
                                包! ');

    //获取接收数据包的开始时间
    dwGetStatusInfoStart:=GetTickCount;
    while (not Terminated) and True do
    begin
        if fMsComm.CDHolding then
        begin
            //cnGetStatusInfoTimeOut
            //dwGetStatusInfoStart, ,dwGetStatusInfoEnd
            //获取接收数据包的结束时间
            dwGetStatusInfoEnd:=GetTickCount;
            if dwGetStatusInfoEnd-dwGetStatusInfoStart>cnGetStatusInfoTimeOut then
            begin //接收信息包超时
                DealError('接收状态信息包超时! ');
                Result:=False;
                break;
            end;
            //接收数据
            faModem.GetInfoFromModem(strReceivedSlice,'Binary');
            faModem.StoreIntostrRecBuffer(strReceivedSlice);
            //接收到完整的数据，进行必要的处理
            if faModem.aPackageReceived.iLen>=cnPackageLen then
            begin
                //校验接收的数据
                Inc(iTotalCommNums); //:integer;总的通信次数
                //function CheckCRC(const aPackageReceived:trPackageReceived):Boolean;
                if CheckCRC(faModem.aPackageReceived) then
                begin
                    //发送 ACK 数据包，如果接收的数据包是正确的，将其存入数据库中
                    aPackageSending.iLen:=faModem.aPackageReceived.iLen;
                    aPackageSending.PackageGeneral:=

```

```

faModem.aPackageReceived.PackageGeneral;
    faModem.SendPackageIntoModem(aPackageSending);
    //写入通信日志
    faModem.ShowModemInfo(fstrPrefixion+'接收到的状态信息包为: ');
    faModem.ShowModemPackage(faModem.aPackageReceived);
    //接收的数据包是正确的, 将其存入数据库中
    if DecodeStatusInfoPackage(faModem.aPackageReceived) then
    begin
        Result:=True;
        //显示状态信息包中的信息
        faModem.ShowStatusPackageInfo;
        faModem.ShowModemInfo(fstrPrefixion+'监测单元'+
fstrPhone+'的提交的状态信息已存入数据库! ');
        StoreStatusPackageIntoDB(fbIsAutoTest,faModem.strModemLog);
    end
    else
        faModem.ShowModemInfo(fstrPrefixion+'监测单元'+fstrPhone+'的提交状态
信息有错! ');
        //清空接收缓冲区, 以接收状态信息包或其他信息
        faModem.ClearRecBuffer;
        //清空接收的数据包
        faModem.aPackageReceived.iLen:=0;
        faModem.ShowModemInfo(fstrPrefixion+'状态信息包接收完毕! ');
        Sleep(cnWaitBeforeHangUp);
    end
    else
    begin
        faModem.ShowModemInfo(fstrPrefixion+'接收的数据包 CRC 错误! ');
        Inc(iCRCBadNums);      //integer; CRC 出错字数
    end;

    DealError("");
    break;      //跳出小循环
end;
end
else      //处理载波丢失
begin
    DealError('接收信息包时载波丢失! ');
    break;

```

```

    end;
    end;          //处理人工测试结束 while True do

end;

end.

```

2. 接收信息包模块分析

监控中心接收信息包的交互图如图 8.18 所示。

当监测单元发现被监测设备的状态异常或者监控中心指定的通信时间已到时，监测单元拨号连接监控中心。监控中心使用两个线程分别监视两个 Modem，平时 Modem 处于自动应答状态，当发现 Modem 被拨通，监视线程自动接收监测单元发过来的数据，校验数据包，如果数据包正确，则发出确认信息包，监测单元接收到确认信息包，如果正确无误，则整个数据包发送过程结束；如果数据包有误，则监控中心抛弃该数据包，同时挂机。监测单元在规定的时间内未接收到确认信息包，挂机，等待一定的时间后，重新拨打监控中心的 Modem。

监测单元到监控中心的 Interaction 图

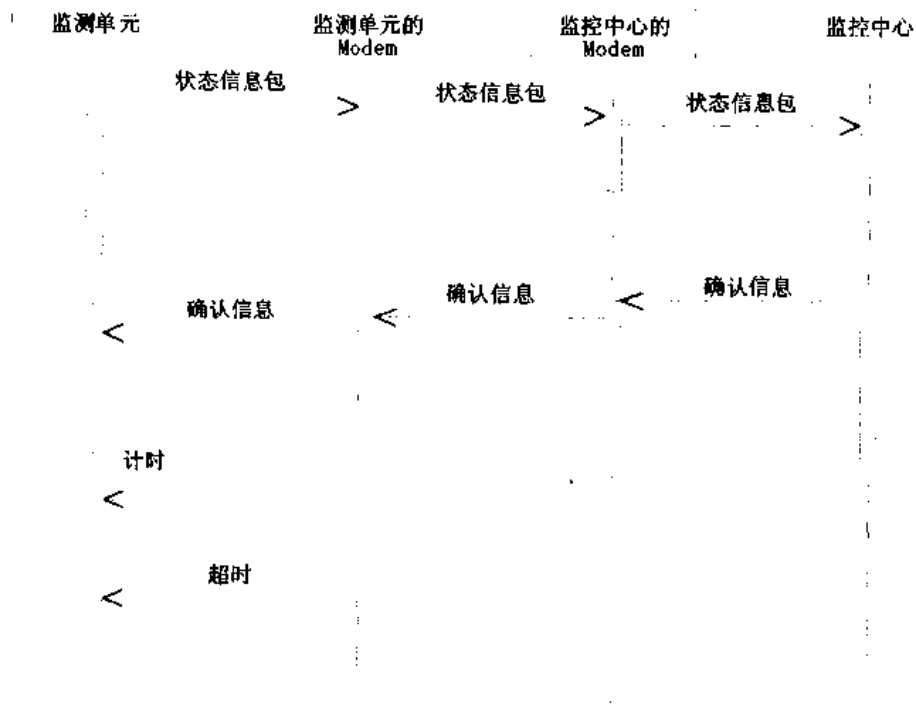


图 8.18 监控中心接收信息包的交互图

接收线程的工作流程图如图 8.19 所示。

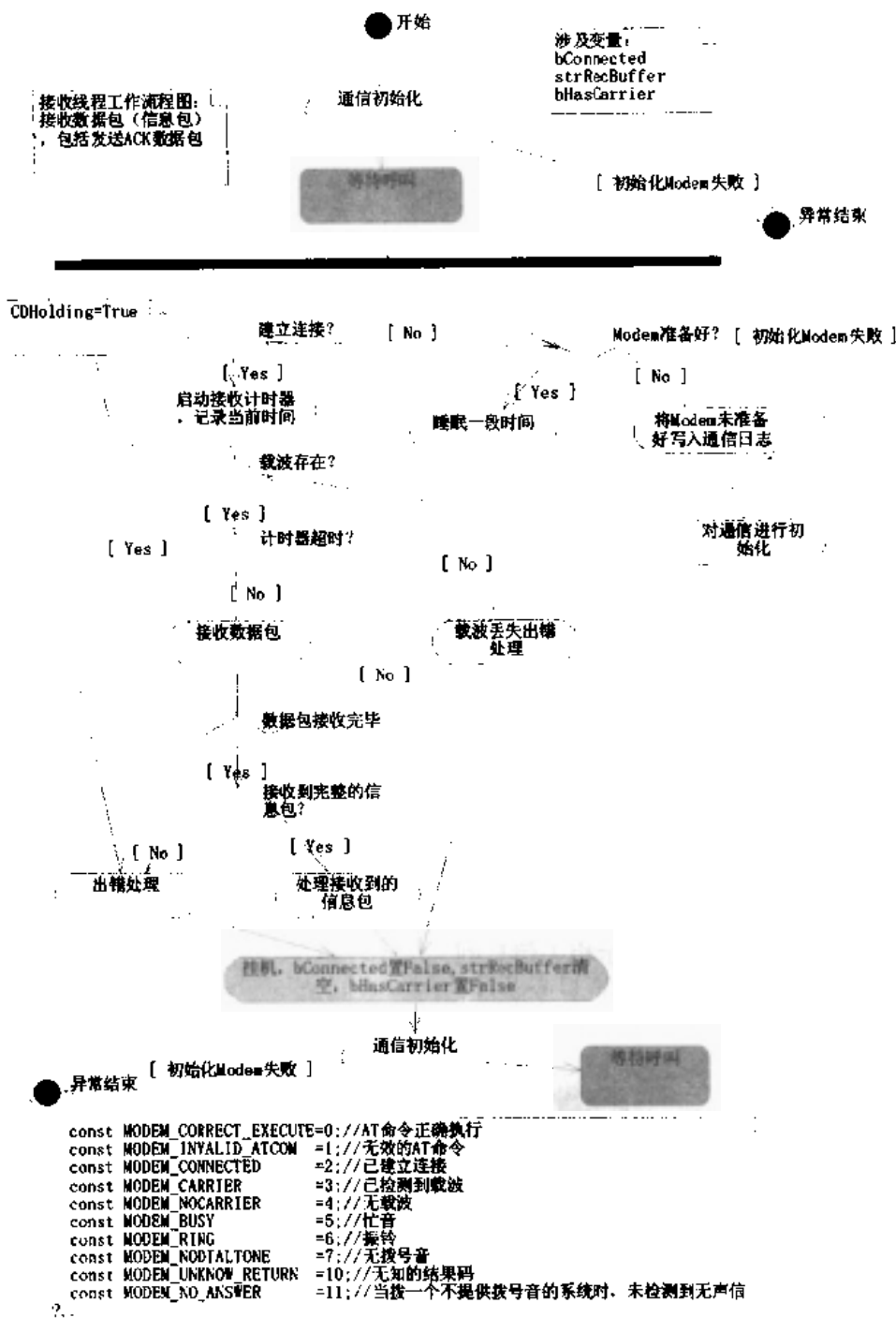


图 8.19 接收线程的工作流程图

其中处理接收到的信息包的流程图如图 8.20 所示。

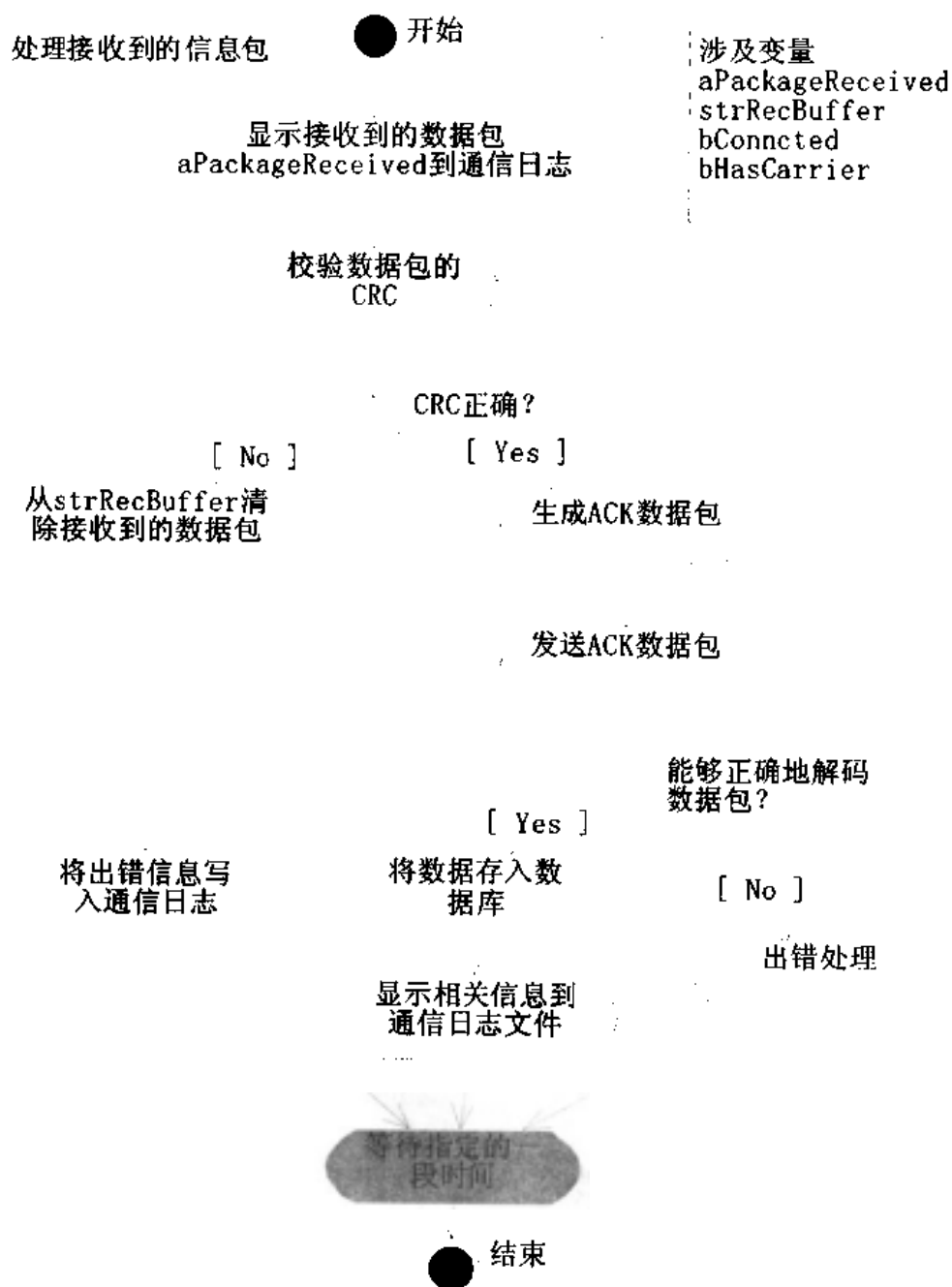


图 8.20 处理接收到的信息包的流程图

单元 uReceiveThread 的类关系图如图 8.21 所示。



图 8.21 单元 uReceiveThread 的类关系图

下面为具体代码:

```
unit uReceiveThread;
```

```

//本线程正常情况下与主线程（主程序）并行运行
//仅当用户需要发送指令到远端的机器上时，才暂时中断运行
//本线程不断地监视 Modem 的状况，当发现有机器想要与本机通信时，Modem 自
//动应答，本线程则接收该机器发送过来的数据包，并给出 ACK 数据包给该机器。
//同时校验接收的数据包，如果数据包是完整、正确的，则将数据存入到数据库中，
//否则，抛弃该数据包
//注意：s38 强制挂机前延迟
//      s10 选择载波掉到挂机之间的延迟
interface
uses

```

```
Classes,OleCtrls,MSCommLib_TLB,
SysUtils,ComCtrls,Windows,cModem,DM,forms,uSendThread;
```

```
const cnReceiveTimeOut=8000;
//接收超时。如果在指定的时间内没能接收到足够的数据，则超时
const cnMaxRecGap=1000*1800;
//cnMaxRecGap 为两次数据接收的最大时间差值， 5*60*1000;该值为 30 分钟
const cnWaitBeforeHangUp=150; //在挂机前的等待时间
```

```
type
```

```
  TReceiveThread = class(TThread)
```

```
  private
```

```
    { Private declarations }
```

```
    fMsComm:TMSCOMM;
```

```
    faModem:TModem;
```

```
    fbIsAutoTest:Boolean;//表示当前工作状态，True 表示自动测试，False 表示人工测试
```

```
    fbInitSuccessful:Boolean; //True 表示初始化 Modem 成功
```

```
    fstrPrefixion:string; //加于 Modem 日志上的前缀
```

```
    procedure EndCommunication;
```

```
    procedure GetIntoWaitForCallingStatus;
```

```
  protected
```

```
    procedure Execute; override;
```

```
    destructor Destroy;override;
```

```
  public
```

```
    constructor Create(aMsCOMM:TMSCOMM;const bIsAutoTest:Boolean);
```

```
    procedure DoTerminate; override;
```

```
    //procedure destroy; override
```

```
  end;
```

```
//记载当前运行的线程数目
```

```
var iReceiveThreadNum:integer;
```

```
    bCom1HasReceiveThread:Boolean;//True 表示串口 1 有对应的接收线程
```

```
    bCom2HasReceiveThread:Boolean;//True 表示串口 2 有对应的接收线程
```

```
implementation
```

```
uses Packages;
```

```
//输入参数: bIsAutoTest 表示当前的工作状态，True 表示自动测试，False 表示人工测试
```

```
constructor TReceiveThread.Create(aMsCOMM:TMSCOMM;const bIsAutoTest:Boolean);
```

```

begin
    inc(iReceiveThreadNum);
    fbIsAutoTest:=bIsAutoTest;
    fMSCOMM:=aMSCOMM;
    fstrPrefixion:='串口'+IntToStr(fMsComm._CommPort)+'接收线程: ';
    // bCom1HasReceiveThread:Boolean, True 表示串口 1 有对应的接收线程
    // bCom2HasReceiveThread:Boolean, True 表示串口 2 有对应的接收线程
    if fMsComm._CommPort=1 then
    begin
        bCom1HasReceiveThread:=True;
    end;
    if fMsComm._CommPort=2 then
    begin
        bCom2HasReceiveThread:=True;
    end;

    faModem:=TModem.Create(fMsComm);
    fbInitSuccessful:=False; //True 表示初始化 Modem 成功
    //fiInitModemAmount:=0; 初始化 Modem 的次数
    inherited create(False);
end;

//中结本线程前的结束工作
procedure TReceiveThread.DoTerminate;
begin
    //中结通信
    // inherited 释放
end;

destructor TReceiveThread.Destroy;
begin
    if fMsComm._CommPort=1 then
        bCom1HasReceiveThread:=False;

    if fMsComm._CommPort=2 then
        bCom2HasReceiveThread:=False;
    // inherited 释放
end;

```

//程序的主体部分

//在发现有远端机器呼叫的情况下，并且双方建立了载波，才启动接收

procedure TReceiveThread.Execute;

var strReceivedSlice:string;

//下面两个变量用于处理接收超时，dwReceiveEnd-dwReceiveStart

//即为接收数据已经用过的时间

fdwReceiveStart:DWord; //开始接收数据的时间点

fdwReceiveEnd:DWord; //结束接收数据的时间点

//下面两个变量用于处理 Modem 工作异常，即如果两个变量的差大于

//cnMaxRecGap，(cnMaxRecGap 为两次数据接收的最大时间差值)

//就认为此时 Modem 工作异常，则重新初始化 Modem

fdwLastReceiveTime:DWord; //上一次接收到数据的时间

fdwCurrentTime:DWord; //当前时间

bIsStatusPackage:Boolean; //True 表示接受的信息包为状态信息包

begin

//放置线程代码

//DMCenter.ADOTWriterMonitors.FieldByName('Communicating').AsBoolean:=True

//1 表示正在与该监测单元通信，fiInitModemAmount:=0

fdwLastReceiveTime:=GetTickCount;

faModem.ShowModemInfo(fstrPrefixion+'初始化接收线程!');

fbInitSuccessful:=FaModem.InitCommunication;

if fbInitSuccessful then

faModem.ShowModemInfo(fstrPrefixion+'接收线程初始化成功!');

else faModem.ShowModemInfo(fstrPrefixion+'接收线程初始化失败!');

//线程主循环

while (fbInitSuccessful) and (not terminated) and true do//大循环，等待呼叫

begin

try

sleep(200);// 检测 Modem 前，休息一会儿

Application.ProcessMessages;//使主线程响应用户的输入

//处理 Modem 工作异常，如果 Modem 在指定的时间内没有收到数据

//则认为 Modem 工作异常，重新初始化 Modem

fdwCurrentTime:=GetTickCount; //获取当前时间

if (fdwCurrentTime-fdwLastReceiveTime>cnMaxRecGap) then

begin

fdwLastReceiveTime:=GetTickCount;//获取当前时间

faModem.ShowModemInfo(fstrPrefixion+'Modem 在'+IntToStr
(cnMaxRecGap div (1000*60))+ '分钟内没有收到数据，重新初始化 Modem!');

//重新初始化 Modem，如果 Modem 无法初始化

```

    if not FaModem.InitCommunication then
        continue; //跳到下一个线程主循环
    end;

    //处理 Modem 工作异常, 如果 Modem 的 DTR 为 Off, 则认为 Modem 异常
    //需要重新初始化 Modem, 如果 Modem 坏、掉电、缺, 则不断地初始化 Modem
    if Not fMsComm.DSRHolding then
        begin
            faModem.ShowModemInfo(fstrPrefixion+
'Modem 当前未准备, 请确认已经连接好 Modem!');
            //重新初始化 Modem, 如果 Modem 无法初始化
            if not FaModem.InitCommunication then
                continue; //跳到下一个线程主循环
            end;
        end;

    if fMsComm.CDHolding then //已经建立连接, 接收远端机器的数据
        begin
            fdwLastReceiveTime:=GetTickCount; //获取最新的数据接收时间
            faModem.ShowModemInfo(fstrPrefixion+'开始接收来自监测单元的状态信息包');
            fdwReceiveStart:=GetTickCount; //开始接收数据的时间点
            while (not terminated) and true do //小循环
                begin
                    if fMsComm.CDHolding then //已经建立连接, 接收远端机器的数据
                        begin
                            //处理接收超时
                            fdwReceiveEnd:=GetTickCount; //结束接收数据的时间点
                            if (fdwReceiveEnd-fdwReceiveStart)>cnReceiveTimeOut then
                                begin
                                    faModem.ShowModemInfo(fstrPrefixion+'接收数据超时! ');
                                    GetIntoWaitForCallingStatus;
                                    break; //跳出小循环
                                end;
                            //接收数据
                            faModem.GetInfoFromModem(strReceivedSlice,'Binary');
                            faModem.StoreIntostrRecBuffer(strReceivedSlice);
                            //接收到完整的数据, 进行必要的处理
                            if faModem.aPackageReceived.iLen>=cnPackageLen then
                                begin
                                    //校验接收的数据

```

```

faModem.ShowModemInfo(fstrPrefixion+'接收到的信息包为: ');
faModem.ShowModemPackage(faModem.aPackageReceived);

Inc(iTotalCommNums);//integer, 总的通信次数
// CheckCRC 函数
if CheckCRC(faModem.aPackageReceived) then
begin
    //发送 ACK 包, 如果接收的包是正确的, 将其存入数据库中
    faModem.ShowModemInfo(fstrPrefixion
+'发送 ACK 数据给单元');
    aPackageSending.iLen:=faModem.aPackageReceived.iLen;
    aPackageSending.PackageGeneral:=
faModem.aPackageReceived.PackageGeneral;
    faModem.ShowModemPackage(aPackageSending);
    faModem.SendPackageIntoModem(aPackageSending);

    //接收的数据包是正确的, 将其存入数据库中
    //function DecodeStoreAPackage 为接收的信息包
    if DecodeStoreAPackage(faModem.aPackageReceived,
fbIsAutoTest,faModem.strModemLog,bIsStatusPackage) then
    begin//如果包能被正确解码, 则将数据存入数据库
        if bIsStatusPackage then//显示状态信息包中的信息
        begin
            //显示信息包的内容
            faModem.ShowStatusPackageInfo;
            //并将结果存入到数据库中
            if StoreStatusPackageIntoDB(fbIsAutoTest,
faModem.strModemLog) then
                begin
                    faModem.ShowModemInfo(fstrPrefixion+
'接收的数据已经存入到数据库中! ');
                end
            else
                faModem.ShowModemInfo(fstrPrefixion+
'接收的数据无法存入到数据库中! ');
            end
        else
            //bIsStatusPackage=False
            faModem.ShowModemInfo(fstrPrefixion+'接收到配置信息');
        end
    end
end

```

```

        end
    else //无法正确解码信息包
        faModem.ShowModemInfo(fstrPrefixion+'接收的数据有错! ');
        Sleep(cnWaitBeforeHangUp);
    end
else
begin
    faModem.ShowModemInfo(fstrPrefixion+'接收的包 CRC 错误');
    Inc(iCRCBadNums);//integer, CRC 出错字数
end;
GetIntoWaitForCallingStatus;
break;//跳出小循环
end;//接收到完整的数据, 进行必要的处理, 处理完毕
end
else//处理载波丢失
begin
    faModem.ShowModemInfo(fstrPrefixion+'载波丢失! ');
    GetIntoWaitForCallingStatus;
    break;//跳出小循环
end;// if fMsComm.CDHolding then 语句结束
end;// while true do 语句结束
sleep(200);
//通信初始化通信信道
fbInitSuccessful:=faModem.InitCommunication;
end;//if fMsComm.CDHolding 语句结束。已经建立连接, 接收远端机器的数据
except
try
    faModem.ShowModemInfo(fstrPrefixion+'???? 线程运行失败, 重新初始化线程! ');
except
end;
end;
end;//while true do 语句结束
//DMCenter.ADOTWriterMonitors.FieldByName('Communicating').AsBoolean:=False
//1 表示正在与该监测单元通信, 挂断电话, 中断本次通信
EndCommunication;
end;
//进入等待呼叫状态
procedure TReceiveThread.GetIntoWaitForCallingStatus;
begin

```



```

faModem.ClearRecBuffer;
faModem.HangUpModem;
end;

//结束通信
procedure TReceiveThread.EndCommunication;
begin
faModem.HangUpModem;//挂断电话, 中断本次通信
Dec(iReceiveThreadNum);
faModem.ShowModemInfo(fstrPrefixion+'终止接收线程! ');
end;

end.

```

8.4 异常处理在程序中的应用

系统中的通信模块对通信中可能出现的异常提供了强大的出错处理机制。比如:

(1) 为了防止监控中心的 Modem 长期工作后, 出现工作异常, 接收线程监测监控中心的 Modem, 如果发现 Modem 在一段时间内未接收到任何数据, 则判定该 Modem 可能出错, 对该 Modem 重新初始化, 同时提示用户该 Modem 可能出错。

源代码如下:

```

//处理 Modem 工作异常, 如 Modem 在指定的时间内没有收到数据
//则认为它工作异常, 重新初始化 Modem
fdwCurrentTime:=GetTickCount;//获取当前时间
if (fdwCurrentTime-fdwLastReceiveTime>cnMaxRecGap) then
begin
fdwLastReceiveTime:=GetTickCount;//获取当前时间
faModem.ShowModemInfo(fstrPrefixion+'Modem 在 '+IntToStr(cnMaxRecGap
div (1000*60))+ '分钟内没有收到数据, 重新初始化 Modem!');

//重新初始化 Modem, 如果 Modem 无法初始化
if not FaModem.InitCommunication then
continue;//跳到下一个线程主循环
end;

```

(2) 在用户发送指令给指定的 Modem 时, 系统将尝试 3 次发送该指令, 知道发送成功。在每一次指令发送过程中, 系统将不断重新初始化 Modem, 直到指定最大初始化次数或者初始化 Modem 成功, 部分源码如下:

```
Inc(fiCallAmount);  
if fiCallAmount>cnCallAmountMax then  
begin  
    Result:=False;  
    DealError('呼叫次数超过指定的标准，终止本次通信！');  
    exit;  
end;
```

读者可以通过笔者提供的通信模块的内部运作流程图发现通信模块的出错处理。由于代码中出错处理机制有些复杂，建议读者参考流程图看源代码，则会容易很多。笔者在设计这段代码时，就是先参考流程图，再编写代码。

本章小结

本章给出了串口通信的一个项目实例。首先介绍了该项目的系统规划设计，对各模块进行了简单的介绍，然后讨论通信协议的设计、通信日志和数据库的设计。考虑到本书主要讲通信编程，本书接下来详细分析与通信相关的代码，最后介绍了系统通信的强大的出错处理。通信模块的出错处理机制有些复杂，建议读者多参考流程图。

第 9 章 RAS 编程

本章主要内容:

- RAS 基本介绍
- 拨号网络配置
- 在程序中实现 RAS 联网

在很多系统中,都涉及到远程数据传输的问题。当然,通过局域网传输数据是最便捷可行的方法。但如果需传输数据的双方之间没有建立局域网连接,在这种情况下,一般有两种解决方案,即使用现成的公共电话网或使用无线微波通信。使用无线微波通信受电磁干扰、天气、距离等因素的限制很大,而且成本高,架设不便,在此不做讨论。而利用公共电话网,通过自己构建拨号服务器和在程序中调用拨号功能,建立客户端和服务端端的 RAS 连接,使得远端用户就如同直接连接在本地局域网上一样访问本地网络资源,方便地实现远程传输数据的功能。

本章首先介绍 RAS 的基本知识,然后介绍使用 RAS 服务所需进行的网络配置,最后讨论如何在 Delphi 程序中调用拨号功能,实现拨号联网和几个 RAS 相关的 Delphi 实例。

9.1 RAS 基本知识

RAS 是微软远程访问服务 (Remote Access Service) 的缩写,它提供了一种通信机制,使得远程用户或移动用户可以通过远程连接方式登录本地 Windows NT 服务器,方便地组成一个广域网。如果远程用户使用 RAS 建立了与本地服务器的连接,就可以通过电话线或其他租用专线实现数据的透明传输,用户可以像本地局域网用户一样访问所有本地网络资源,只是速度要低于局域网的访问速度。

一个普通类型的 RAS 结构如图 9.1 所示。在这个小型拨号网络中,Windows NT RAS 服务器带有两个外部调制解调器,Windows 95 客户和 Windows NT 工作站及带有 PC 卡调制解调器的便携机提供了 DUN (Dial-Up NetWork 拨号网络) 服务。远端用户在 Windows 95 客户端或 Windows NT 工作站拨号与 Windows NT RAS 服务器建立 RAS 网络连接,即可访问 RAS 服务器的本地网络资源。

RAS 基于客户/服务器机制,服务器端和客户端都应安装相应的网络适配器和软件。拨入网是 RAS 的客户部分,RAS 的服务器部分实现远程访问管理。通常情况下,RAS 连接可用电话线和一个 Modem 实现,此外,RAS 还支持 X.25、ISDN、VPN 等连接方式。在 Windows NT Server 中,一台 RAS 服务器最多可支持 256 个并发会话 (Session),这些会话可以是对外的也可以是从外部来的。RAS 在网络层支持 IPX、TCP/IP 和 NetBEUI,在传输层支持点对点

协议 (PPP) 和串行 Internet 协议 (SLIP)。RAS 连接可使用网络层协议的各种组合, 但传输层协议必须为 PPP 或 SLIP。

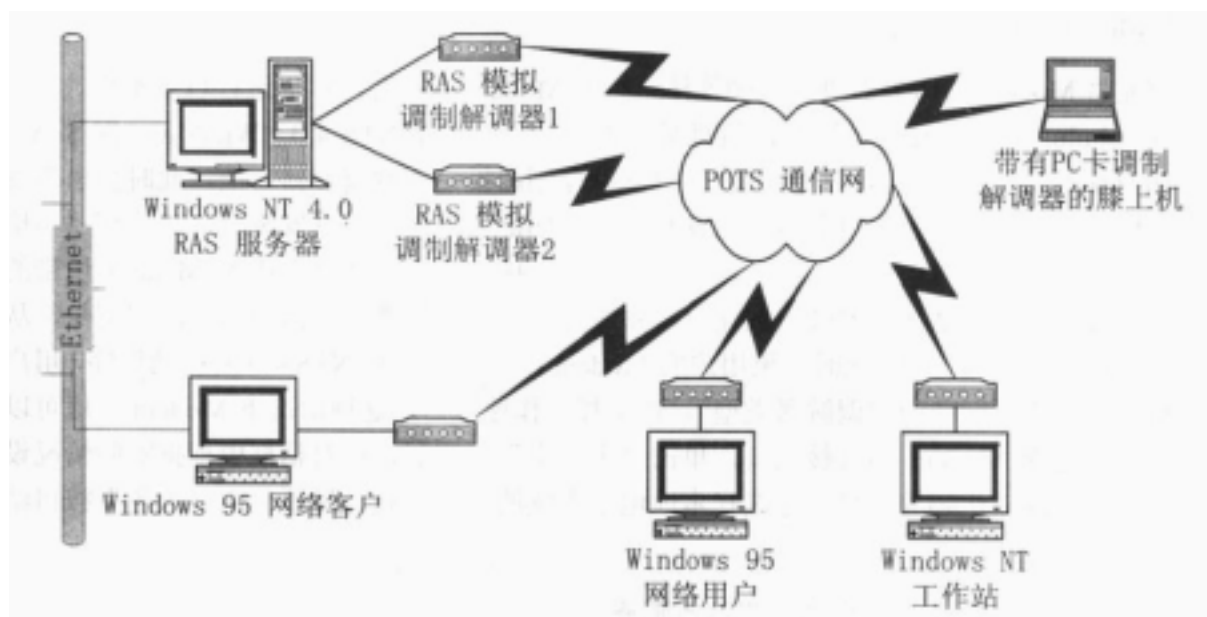


图 9.1 带有用模拟 Modem 通信的 Windows NT 和 Windows 95 RAS 客户服务器的小型拨号网

在具体应用中, 如果远程用户不是很多, 远程存取的数据量不是十分庞大, 不需要连续高速连接时, 可以不必增加投资去配置远程访问的硬件设备及租用专线, 最简单也最经济的方法是利用计算机内置的串行通信端口, 通过调制解调器 (Modem) 和常规的电话线进行连接。

注意: Windows NT Server 可支持 255 个同时连接的 RAS, 而 Windows NT Workstation 只支持一个客户端的 RAS 连接。Windows 95 支持单拨号客户端 RAS 连接。路由与远程访问服务 RAS (Remote Access Service) 是 Windows 2000 的一个重要功能, 它是 Windows NT 4.0 下的远程访问服务 RAS 和路由信息协议 RIP (Routing Information Protocol) 的结合。

9.2 拨号网络的配置

用户要使用 RAS 服务, 无论是直接建立 RAS 连接, 还是在程序中调用 RAS 函数, 都必须先配置拨号网络。拨号网络的配置包括拨号客户端的配置和拨号服务器端的配置。

9.2.1 Windows NT 4.0 拨号服务器配置

拨号服务器端主机必须安装 Windows NT 4.0 操作系统或 Windows 2000 操作系统并提供远程访问, 安装 Windows NT 4.0 远程访问服务 (RAS) 包括下面的步骤。

(1) 安装 Modem。

(2) 为拨号网络配置 RAS。

(3) 创建远程访问服务用户账号并设置访问权限。

1. Modem 的安装和设置

首先将 Modem 与服务器进行正确连接, 打开 Modem 的电源, 然后进行以下操作。

以 Administrator (服务器默认的管理员身份) 的身份登录 NTS 4.0 (Windows NT Server Version 4.0 的简称) 服务器, 打开“我的电脑→控制面板→调制解调器”项, 此时会出现安装对话框。如果不想让系统自动检测已安装的 Modem 而直接从磁盘安装时, 可选择“不检测调制解调器; 将从清单中选定”一项, 否则不选。接下来在驱动器中插入 Modem 所带的驱动程序盘, 单击“从磁盘安装”按钮。在系统提示下, 选择调制调器的制造商和型号, 从磁盘安装对应的驱动程序。此时如果用户的 Modem 不能提供支持 NTS 4.0 的驱动程序, 用户也可将其设置成“标准调制解调器类型”, 再选择工作速度后, 这种情况下 Modem 一般可以正常工作。选择 Modem 的连接端口, 单击“下一步”按钮后, 在对话框中根据实际情况设定有关国家、区域代码等信息, 并选择本地电话系统的工作方式, 最后单击“关闭”按钮结束。

2. 在 NTS 4.0 服务器上安装远程访问服务

当 Modem 安装好后, 可按以下的步骤安装远程访问服务。

首先依次打开“我的电脑→控制面板→网络→服务”项, 在对话框的“网络服务”下拉列表中选择“远程访问服务”一项, 单击“确定”按钮。输入 NTS 4.0 安装文件的路径后单击“确定”按钮, 系统开始从安装盘复制所需的文件。文件复制结束后, 出现对话框, 在“远程访问服务可用设备”列表中显示了已安装的 Modem 类型。单击“确定”按钮后, 单击“配置”按钮, 在接下来出现的对话框中可设置“端口用法”。

☐ “只能拨出”表示此台计算机只能做为远程访问服务工作站。

☐ “只能接收”表示可将此台计算机设置成为远程访问服务服务器。

☐ “拨出和接收”表示这台计算机在不同的时间内即可作为远程访问服务服务器, 也可作为远程访问服务工作站。

接下来返回安装远程访问对话框, 单击“网络”按钮, 将出现网络配置对话框。在“拨出协议”下方提供了 3 种可供计算机在拨出时所使用的通信协议 NetBEUI、TCP/IP 和 IPX。在以下的设置中, 可以根据远程访问服务工作站所安装操作系统的不同来确定, 不过在 3 种通信协议中 NetBEUI 占用计算机内存最小、速度最快, 且支持所有常用的操作系统, 同时协议的配置也很简单, 所以建议使用 NetBEUI 通信协议。值得指出的是, 用户可以同时选择 3 种协议, 以满足不同的远程访问服务工作站远程登录的需要。在“允许远程客户运行”下方也提供了 3 种与拨出相同的协议, 可根据需要来选择。为了有利于数据传输的安全性, 防止数据传输时被窃取, 在“加密设置”下方提供了 3 种加密方法, 可视具体需要选择。完成上述设置后, 重新启动系统就可以了。

3. 创建远程访问服务用户账号并设置访问权限

远程访问服务安装完毕后, 如果远程访问服务工作站的用户账号还没有建立, 必须先利用“开始→程序→管理工具(公用)→域用户管理器”项创建用户账号。之后再选择“开始

→程序→管理工具（公用）→“远程访问系统管理”命令打开创建对话框。在“服务器”列表中选择已建立的远程访问服务服务器，然后选择“用户”菜单下的“权限”项，将出现一个新的对话框。在“用户”列表框中选择一个用户名，然后选择“赋予用户拨入的权限”一项，该用户便被设置为远程访问服务用户，拥有远程访问网络的权力。其他用户可用同样的方法设置。如果要将“用户”列表中的所有用户设置成为远程访问服务用户，可单击对话框中的“全部给予”按钮。每个远程访问服务用户可使用3种“回拨”方式与远程访问服务服务器通信，这3种回拨方式的含义如下。

□ 不回拨：用户可直接拨入，只要远程访问服务用户的账号和密码正确，就可连入网络。这是系统缺省的选项，但通信中的安全性较差。

□ 由拨号者设置：这种工作方式是当远程访问服务用户拨号登录后，远程访问服务服务器要求用户输入回拨的电话号码，当用户输入并确认后，系统自动切断远程连接，然后再由远程访问服务服务器根据所接收到的回拨号码建立与远程访问服务工作站的通信。

□ 预设到当某一远程访问服务：用户选择了此功能后，需在“预设到”后面的文本框中输入该远程用户的回拨电话号码，单击“确定”按钮后系统便将此号码保存在远程访问服务服务器中。当远程访问服务用户使用预设的号码登录网络时，远程访问服务服务器接到拨号后可回拨给工作站，但该用户利用另一条电话线连接时将被拒绝。这是3种方式中最安全的一种。

9.2.2 Windows 2000 远程访问服务器的配置

微软在 Windows 2000 Server 内提供的远程网络服务，较之以前版本有不少新特点：首先，远程访问服务可以和 Windows 2000 Active Directory 集成在一起，作为 Windows 2000 域的一部分。其次，提供更可靠的安全控制（如 MS-CHAP V2、可扩展的身份验证协议、RADIUS 客户、账户锁定等），更方便的管理手段（如路由和远程访问管理、带宽分配协议、远程访问策略），更多种的远程服务协议（如支持第二层隧道协议、支持 AppleTalk Macintosh 客户机远程访问、转发 IP 多播通信）。

安装 Windows 2000 服务器时，“路由和远程访问服务”组件已经自动安装，但是该服务处于非激活状态，需要手工启用和配置。下面以服务器用调制解调器拨号连接方式为例，具体介绍远程访问服务的安装与设置，其他方式的远程接入设备的安装请参阅相关资料。

（1）安装调制解调器

首先确保调制解调器正确连接到电话线和计算机上。一般符合即插即用的调制解调器均能被 Windows 2000 检测到并自动安装。如果调制解调器不能自动安装，使用“控制面板”中的“电话和调制解调器选项”的“调制解调器”标签手动安装调制解调器。

（2）配置路由和远程访问属性

执行“管理工具→路由和远程访问”命令，屏幕将出现“路由和远程访问”窗口。选中该窗口左边“树”中的“本地”项，然后选“操作”选单中的“配置并启用路由和远程访问”项，屏幕出现“路由和远程访问服务安装向导”欢迎窗口，单击“下一步”按钮。

（3）根据向导选择“远程访问服务器”，单击“下一步”按钮，选择现有的或者添加一个远程客户协议（如 TCP/IP），单击“下一步”按钮。

（4）如果在步骤3选择 TCP/IP 协议，向导将会要求读者指出如何对远程客户分配 IP 地

址，选择“来自一个指定的地址范围”，也可以选择“自动”让 DHCP 服务器或者远程访问服务器为读者自动生成，单击“下一步”按钮。

(5) 如在上一步选择了指定远程客户的 IP 地址，那么将在这一步通过“新建”按钮设置远程客户静态的 IP 地址（如：起始 IP 地址：192.168.181.2，结束 IP 地址：192.168.181.6），然后单击“下一步”按钮。

(6) 选择不使用 RADIUS，单击“下一步”按钮，最后单击“完成”按钮，至此，激活了路由和远程访问服务。

现有的网络用户，需要经远程访问授权后，才可以在远地访问远程访问服务器及网络。方法为通过管理工具“Active Directory 用户和计算机”（如远程访问服务器是独立服务器用“计算机管理”），选择用户后单击鼠标右键，在弹出的快捷键菜单中选择“属性”项，打开“拨入”标签。可以设置远程访问权限、回拨属性等。

9.2.3 拨号客户端主机的配置

(1) 安装 Modem

在 Windows 95/98 上安装 Modem 的方法与在远程访问服务服务器上的安装方法基本相同，而且因 Windows 98 支持 PnP 功能，将使安装过程更为简单。

(2) 安装拨号网络

在 Windows 98 下拨号网络是缺省安装的。Windows 95 下如果还未安装拨号网络，则按以下过程，即“开始→设置→控制面板→添加/删除程序→Windows 安装程序→通信”中选择“拨号网络”，开始安装。

(3) 建立 Windows 95/98 的拨号连接

选择“我的电脑→拨号网络→新建连接”项，将出现一个对话框，在此用户可以输入要拨入的远程访问服务服务器名称。然后单击“下一步”按钮，在对话框中输入远程访问服务服务器方的区号、电话号码、国家与地区代码，输入在服务器端新建的用户名和密码，单击“下一步”按钮完成设置。

(4) 设置拨号连接项的属性

右键单击新建的拨号连接项，在弹出的快捷菜单中选择“属性”。在出现的对话框中选择“服务器类型”标签，并在“拨号网络服务器类型”下拉列表中选择 PPP 协议。其中 PPP 是 Windows NT 远程访问服务服务器在进行拨入和拨出时经常使用的一个点对点通信协议，它很适合电话线的远程连接，目前在连入因特网时一般也使用此协议。然后在“所允许的网络协议”下方选择必要的通信协议，如：TCP/IP 协议，IPX/SPX 兼容协议，NetBEUI 协议。

这样就完成了拨号网络的配置，接下来将介绍如何在程序中调用拨号功能，建立客户端、服务器之间的远程连接，从而实现远程数据传输。

9.3 在程序中实现 RAS

远程用户利用电话线通过远程访问服务器连接到本地网络。一旦连接成功后，电话线就

是透明的，用户可以访问所有的网络资源，RAS 使调制解调器象网卡一样的工作。为了获得这种功能，首先，需要在服务器上安装 Windows NT，然后安装调制解调器（标准调制解调器），再在 NT 的网络配置中安装远程访问服务，配置各种参数。在客户端（Windows 95/98 操作系统）安装拨号网络，然后建立连接项。在配置好了 RAS 客户端和拨入服务器端后，就可以在程序中调用 RAS 的 API 函数了。

在程序中调用 RAS 的 API 函数之前，首先简单地介绍一下 RAS 的 API 函数。

9.3.1 RAS 的 API 函数简介

RAS 的一些公共函数如表 9.1 所示。

表 9.1 RAS 的 API 函数

RAS 函数	执行的动作
RASADFunc	应用程序定义的回调函数
RasConnectionNotification	当创建/终止一个 RAS 连接时，指定一种设置系统为有信号状态的对象
RasDial	在一个 RAS 客户和一个 RAS 服务器间建立一个 RAS 连接
RasDialFunc2	应用程序定义的回调函数，在状态改变时由 RasDial 激活
RasDialDlg	用一种特定的电话簿入口建立一个 RAS 连接
RasEntryDlg	操纵电话簿入口的特征页
RasEnumConnections	列出活跃的 RAS 连接，包括句柄和电话簿
RasGetConnectStatus	当前 RAS 连接的状态
RasGetEntryDialParams	从为某电话簿入口的最后一个成功调用检索连接信息
RasGetErrorString	将 RAS 错误码转化为错误字符串
RasMonitorDlg	描述 RAS 连接状态的特征页
RasHangUp	终止一个 RAS 连接

9.3.2 使用动态链接库实现 RAS 的函数调用

在 Windows 家族中，Windows NT 中可以安装远程访问服务（RAS），Windows 95/98 提供了拨号上网的客户端功能。但这些操作系统都没有提供实现拨号网络功能的 API 函数。在 Windows 中，拨号网络的调用都是在动态链接库 RASAPI32.DLL 中实现的。

在程序中，系统初始化时就要将该动态链接库载入内存，然后再载入需要调用的函数。具体调用方法如下。

```
RasLib: Thandle;
//定义动态链接库的句柄
RasLib=LoadLibrary('RASAPI32.DLL');
//载入动态链接库
```


在这个 DLL 中，系统用到的函数包括：拨号函数 `RasDial`、挂断函数 `RasHangUp`、连接状态查询函数 `RasGetConnectStatus`；还有一些和拨号连接项管理有关的函数，`RasGetEntryDialParams` 函数、`RasSetEntryDialParams` 函数、`RasEditPhonebookEntry` 函数、`RasCreatePhonebookEntry` 函数。

要在程序中实现拨号上网功能，其大体过程如下：

1. 利用 Modem 拨号进行连接

利用 Modem 拨号进行连接，首先使用 `RasDial` 函数。

`RasDial` 的声明如下：

```
Function RasDial(
    lpRasDialExtensions : PRASDIALEXTENSIONS ;
    //在 WINDOWS 9x 下无用，可设置为 NIL
    lpzPhonebook: PChar;      //在 Windows 9x 下无用，可设置为 NIL
    lpRasDialParams : PRASDIALPARAMS; // 拨号参数，类型指针
    dwNotifierType : DWORD;      // 消息通知方式
    lpvNotifier: DWORD;          // 消息处理事件
    var rasConn: HRASConn        // 返回成功连接的连接句柄
): DWORD; stdcall;
```

参数说明：

`lpRasDialExtensions` 和 `lpzPhonebook`：仅在 Windows NT 下有效，在 Windows 95/98 下，这两个参数被忽略。

`lpRasDialParams`：这个参数很重要，它指向一个 `RASDIALPARAMS` 结构，该结构包含以下几个成员。

`dwSize`：应设定为 `sizeof (RASDIALPARAMS)`。

`szEntryName` 和 `szPhoneNumber`：这两个参数有联系，`szEntryName` 可以指定要建立的连接，例如“我的连接”是处理用户已经在“拨号网络”里建立的连接。这时，Modem 将拨打用户在“我的连接”中设定的 ISP 号码，此时 `szPhoneNumber` 成员设为空字符串“”即可；如果用户要在程序中自行指定要拨打的 ISP 号码的话，`szEntryName` 可以设定为空字符串“”，此时应设置 `szPhoneNumber` 为读者的 ISP 号码（169，163 等），对于用 201 电话卡来上网的情况，可以设为“201,账号，密码#，ISP 号码#”（其中“，”表示停顿一段时间（以等待确认账号，密码等），用户可以根据自己所在位置的线路状况自行调节。

`SzCallBackNumber`，`szDomain`：设为空串“”即可。

`SzUserName`，`szPassword`：登录用户名和密码。如 169 用户的用户名为 `guest`，密码为 `guest`；163 用户的用户名为 `163`，密码为 `163`。

其他成员不必设置。

`DwNotifierType`：指定是由窗口还是由回调函数来处理确认消息。通过确认消息用户可以得到 `RasDial` 过程的当前状态。如“正在打开段口”、“正在验证用户名和密码”等。也可设为 `NULL`。

`dwNotifier`：指定处理确认消息的窗口或回调函数。也可设为 `NULL`。

LphRasConn: 指向一个类型为 HRASCONN 的变量。在调用 RasDial 前必须指定为 NULL, RasDial 若成功返回, 则将 RAS 连接的句柄存放于它所指向的变量中。用户也可以通过此句柄来断开连接。只要在程序中适当位置调用 RasDial 函数即可建立连接。

返回值:

成功则返回值为 0, 否则返回错误代码。程序中可调用 RasGetErrorString 函数将 RAS 错误码转换成错误字符串。

```
RasGetErrorString: Function (
    ErrorCode: DWord;    //错误码
    szErrorString: Pchar; //存放错误字符串的缓存
    BufSize: Dword;      //存放错误字符串的缓存的大小
): LongInt; stdcall;
```

2. 获得拨号过程的当前状态

通过调用 RasGetConnectStatus 函数, 获得拨号过程的当前状态。

```
RasGetConnectStatus: Function (
    RASConn: hrasConn;      // Internet 的远程访问连接的句柄
    RASConnStatus: PRASConnStatus //接收状态数据的缓冲器
): LongInt; stdcall;
```

参数说明:

RASConn: 通过 RasDial 建立的拨号连接的句柄。

RASConnStatus: 存放当前拨号连接的状态, 是一个 TRASConnStatus 结构的指针变量。

PRASConnStatus = ^TRASConnStatus;

```
TRASConnStatus = Record
    dwSize: LongInt;
    rasConnstate: Word;
    dwError: LongInt;
    szDeviceType: Array[0..RAS_MaxDeviceType] Of Char;
    szDeviceName: Array[0..RAS_MaxDeviceName] Of Char;
    szPhoneNumber: Array[0..RAS_MaxPhoneNumber] Of Char;
end;
```

3. 断开连接

断开拨号连接的 RasAPI32 函数为 RasHangUp, 其函数原型为:

```
RasHangUp(
    HRASCONN: DWORD
): DWORD; stdcall;
```

参数说明:

HRASCONN: DWORD 要挂断的拨号连接的句柄。

返回值:

函数的返回值为 0 表示执行成功, 不为 0 执行失败。

断开过程如下:

```
if m_hRasConn <> Nil then
begin
```

```
    RasHangUp(m_hRasConn);
    m_hRasConn:= Nil;
    m_OnDial:=TRUE;
    Sleep(2000);
```

```
end;
```

注意: 以上代码中的 Sleep 函数是必需的。需要一定时间来断开连接。如果不等待一段时间, 计算机有可能无法正常关闭端口。导致下一次无法拨号, 只有重新启动 Windows 才能解决。

要预防此问题也可以调用 RasGetConnectStatus 函数, 方法如下:

```
RASConnStatus: TRASConnStatus;
```

```
while RasGetConnectStatus (m_hRasConn, @rStatus) <> ERROR_INVALID_HANDLE do
begin
```

```
    Sleep (10) ;
```

```
end;
```

通常, 可以调用 RasEnumConnection 函数来得到当前连接的句柄。

RasEnumConnections 函数原型为:

```
RasEnumConnections(LPRASCONN lprasconn,
    LPDWORD lpcb,
    LPDWORD lpcConnections
): DWORD;stdcall;
```

参数说明:

lprasconn: 接收活动连接的缓冲区的指针。

lpcb: 接收缓冲区的字节大小。

lpcConnections: 实际的活动连接数。

返回值:

函数的返回值为 0 表示执行成功, 不为 0 执行失败。

这样, 通过调用动态连接库中的 API 函数, 一个实现基本拨号上网功能的程序就完成了。

9.3.3 在 Delphi 程序中拨号上网

在 Windows 9x 下, 如果安装了拨号网络, 则在 Windows 系统的系统目录 System 下将有两个拨号网络管理程序库 RasApi32.DLL 和 RasApi16.DLL, 可利用其中的函数来创建、修改拨号连接, 并利用指定的拨号连接进行拨号上网。

1. 新建拨号连接

当 Windows 9x 系统中已经建立了拨号连接, 则可利用现成的拨号连接。如果没有拨号连接, 则需要新建一个拨号连接。RasAPI 中提供了相应的函数, 其函数名为

RasCreatePhonebookEntryA, 函数原型为:

```
function RasCreatePhonebookEntryA( hwnd : THandle; lpszPhonebook: pchar ) :
DWORD;stdcall; //位于 interface 部分
```

```
function RasCreatePhonebookEntryA; external 'Rasapi32.dll';
```

//位于 implementation 部分

参数说明:

hwnd (THandle): 新建拨号连接窗口的父窗口的句柄, 可以为 TForm 的 Handle, 为 Nil 表示 Windows 桌面 (Desktop)。

lpszPhonebook (pchar): 电话本名称, 在 Windows 9x 下无作用, 可设为空字符串。

函数返回值: 0 表示执行成功, 否则为错误。

下面是一个新建拨号连接的例子。

//新建拨号连接

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
    dwResult : DWORD;
```

```
begin
```

```
    //在当前窗口中新建拨号连接
```

```
    dwResult := RasCreatePhonebookEntryA( handle, " );
```

```
    if dwResult = 0 then
```

```
        memo1.lines.add('新建拨号连接成功!')
```

```
    else
```

```
        memo1.lines.add('新建拨号连接失败!')
```

```
end;
```

2. 修改指定拨号连接的属性

如果用户需要修改拨号连接的属性如电话号码、国家及区号、连接方式、服务器类型等, 可以用 RasAPI 函数来实现, 其函数名为 RasEditPhonebookEntryA, 函数原型为:

```
function RasEditPhonebookEntryA( hwnd : THandle; lpszPhonebook: pchar;
```

```
lpszEntryName: pchar ) : DWORD;stdcall; //位于 interface 部分
```

```
function RasEditPhonebookEntryA; external 'Rasapi32.dll'; //位于 implementation 部分
```

参数:

hwnd (THandle): 新建拨号连接窗口的父窗口的句柄, 可以设为 TForm 的 Handle, 为 Nil 表示 Windows 桌面 (Desktop)。

lpszPhonebook (pchar): 电话本名称, 在 Windows 9x 下无作用, 可设为空字符串。

lpszEntryName (pchar): 要修改的拨号连接的名称, 如“163”、“169”等。

函数返回值: 0 表示执行成功, 否则为错误。

下面是一个修改指定拨号连接属性的例子。

//修改指定拨号连接属性

```
procedure TForm1.Button2Click(Sender: TObject);
```

```

var
    dwResult : DWORD;
    strDialName : string;
begin
    strDialName := '163'; //拨号连接的名称设为 163
    //在当前窗口中指定修改拨号连接的属性
    dwResult := RasEditPhonebookEntryA( handle, ", PChar( strDialName ) );
    if dwResult = 0 then
        memo1.lines.add('修改拨号连接' + strDialName + '成功!')
    else
        memo1.lines.add('修改拨号连接' + strDialName + '失败!');
end;

```

3. 获取当前系统中可用的拨号连接名称

为了让用户选择使用拨号连接进行拨号，需要获取系统中已建立的拨号连接的名称。在建立了拨号连接后，Windows 9x 将拨号连接的名称和属性写在了注册表中，可以从注册表中获取当前系统中可用的拨号连接名称及在 Internet Explorer 中设置的默认连接名称。

在注册表的 HKEY_USERS\Default\RemoteAccess\Addresses 下，列出了已经在拨号网络中建立了的拨号连接的名称及其属性设置，其中各项目的名称即为可用的拨号连接的名称，各项目的值即为各拨号连接的属性设置。只要读出各项目的名称即可获取当前系统中可用的拨号连接名称。

如果在 Internet Explorer 中设置了默认连接名称（工具→Internet 选项→连接→始终拨打默认连接），则在注册表的 HKEY_USERS\Default\RemoteAccess 下，有一个字符串类型的键值，键值名 InternetProfile，其值即为 Internet Explorer 中设置的默认连接名称。

下面是一个获取当前系统中可用的拨号连接名称的例子。

//注意在 Uses 中增加 Registry 单元，用于操作注册表

//获取当前系统中可用的拨号连接名称

```

procedure TForm1.Button3Click(Sender: TObject);
var
    registryTemp : TRegistry;
    stringsTemp : TStringlist;
    intIndex : integer;
begin
    registryTemp := TRegistry.Create;
    stringsTemp := TStringlist.Create;
    with registryTemp do
    begin
        RootKey := HKEY_USERS; //根键设置为 HKEY_USERS
        //如果存在子键.Default\RemoteAccess\Addresses

```

```

    if OpenKey('.Default\RemoteAccess\Addresses',false) then
        GetValueNames( stringsTemp );
        //读出各项目的名称，即拨号连接名称
    CloseKey;
end;
//当前系统中可用的拨号连接
memo1.lines.add('*****当前系统中有'+IntToStr( stringsTemp.count)
                +'个可用的拨号连接如下*****');
for intIndex := 0 to stringsTemp.count - 1 do
    memo1.lines.add( stringsTemp.strings[ intIndex ] );
//列出 Internet Explorer 中设置的默认连接名称
if registryTemp.OpenKey('.Default\RemoteAccess',false) then
    memo1.lines.add( 'Internet Explorer 中设置的默认连接名称为' +
                    registryTemp.ReadString('InternetProfile') );
//释放内存
registryTemp.free;
stringsTemp.free;
end;

```

4. 用指定的拨号连接拨号

以上的三个工作的目的就是为了拨号上网，现在就来看看如果用指定的拨号连接拨号上网。最好的方法就是调用 Windows 9x 的拨号网络服务了，就是运行 Windows 9x 下的现成程序。在 Delphi 程序中可以用如下代码实现拨号上网：

```
winexec('rundll32.exe maui.dll,RnaDial 163',SW_SHOWNORMAL);
```

其中字符串中的最后一个参数“163”为拨号连接的名称。

下面是一个用指定的拨号连接拨号上网的例子。

//用指定的拨号连接拨号上网

```

procedure TForm1.Button4Click(Sender: TObject);
var
    strDialName : string;
begin
    strDialName := '163';//拨号连接的名称设为 163
    memo1.lines.add('用拨号连接'+ strDialName +'实现拨号上网');
    winexec(PChar('rundll32.exe maui.dll,RnaDial'+strDialName),SW_SHOWNORMAL);
end;

```

9.3.4 断开 Internet 连接的程序

下面给出一个程序，展示如何调用 RAS（远程访问服务）系列函数断开指定的拨号连接 UnitDemo 单元的窗体如图 9.2 所示：



图 9.2 UnitDemo 单元的窗体

1. 程序主体

程序代码如下：

```
unit UnitDemo;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, DdeMan;
type
  TForm1 = class(TForm)
    ButtonDisconnect: TButton;
    Label1: TLabel;
    procedure ButtonDisconnectClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
  uses rasapi;
  {$R *.DFM}

  procedure TForm1.ButtonDisconnectClick(Sender: TObject);
  var
    bufsize: Cardinal;
    numEntries: Cardinal;
    x: Integer;
    entries: Array[1..10] of TRasConn; // 拨号连接数组（假设最多有 10 个活动的拨号连接）
    stat: TRasConnStatus;
```

```

    hRas: HRasConn;
    conexion:pRasConn;
begin
    entries[1].dwSize := SizeOf(TRasConn);
    bufsize := SizeOf(TRasConn) * 10;
    FillChar(stat, Sizeof(TRasConnStatus), 0);
    stat.dwSize := Sizeof(TRasConnStatus);
    if RasEnumConnections(@entries[1], bufsize, numEntries) = 0 then
    begin
        if numEntries > 0 then
            with entries[1] do
            begin
                conexion:=Pointer(rasconn);
                hRas := Longint(conexion);
                if RasHangUp(hRas) <> 0 then
                    ShowMessage('断开失败');
                end
            else
                ShowMessage('断开成功');
            end;
        end;
    end;
end.

```

2. 单元 rasapi

程序中使用了 RAS 系列函数，下面给出程序中用到的 RAS 函数，并组成一个可独立调用的单元库。

```

unit rasapi;
interface
    Uses WinTypes, WinProcs, Dialogs;
Const
    UNLEN = 256;
    PWLEN = 256;
    DNLEN = 15;
    RAS_MaxEntryName    = 256; //连接名称最大长度
    RAS_MaxDeviceName   = 128; //设备名称最大长度
    RAS_MaxDeviceType   = 16;  //设备类型名称长度
    // RAS_MaxParamKey   = 32;
    //RAS_MaxParamValue   = 128;
    RAS_MaxPhoneNumber  = 128; //电话号码的最大长度

```



```

RAS_MaxCallbackNumber = RAS_MaxPhoneNumber;
Type
//UINT = Word;
PHRASConn = ^HRASConn;
HRASConn = DWORD;
Type
//确认一活动的拨号连接（为 RasConnectEnum 函数使用）
PRASConn = ^TRASConn;           //拨号连接的句柄类型
TRASConn = record
    dwSize: DWORD;                //该结构类型所占内存的字节大小
    rasConn: HRASConn;            //活动连接的句柄
    szEntryName: Array[0..RAS_MaxEntryName] Of Char;
    szDeviceType : Array[0..RAS_MaxDeviceType] Of Char;
    szDeviceName : Array [0..RAS_MaxDeviceName] of char;
end;
PRASConnStatus = ^TRASConnStatus;
TRASConnStatus = Record
    dwSize: LongInt;
    rasConnstate: Word;
    dwError: LongInt;
    szDeviceType: Array[0..RAS_MaxDeviceType] Of Char;
    szDeviceName: Array[0..RAS_MaxDeviceName] Of Char;
end;
PRASDIALEXTENSIONS= ^TRASDIALEXTENSIONS;
TRASDIALEXTENSIONS= Record
    dwSize: DWORD;
    dwfOptions: DWORD;
    hwndParent: HWND;
    reserved: DWORD;
end;
PRASDialParams = ^TRASDialParams;
TRASDialParams = Record
    dwSize: DWORD;
    szEntryName: Array[0..RAS_MaxEntryName] Of Char;
    szPhoneNumber: Array[0..RAS_MaxPhoneNumber] Of Char;
    szCallbackNumber: Array[0..RAS_MaxCallbackNumber] Of Char;
    szUserName: Array[0..UNLEN] Of Char;
    szPassword: Array[0..PWLEN] Of Char;
    szDomain: Array[0..DNLEN] Of Char;

```

```

end;
PRASEntryName = ^TRASEntryName;
TRASEntryName = Record
    dwSize: LongInt;
    szEntryName: Array[0..RAS_MaxEntryName] Of Char;
// Reserved: Byte;
end;
//拨号函数
Function RasDial(
    lpRasDialExtensions: PRASDIALEXTENSIONS ; // Window 9x 下无用, 可设置为 NIL
    lpszPhonebook: PChar; //在 Window 9x 下无用, 可设置为 NIL
    lpRasDialParams: PRASDIALPARAMS; // 拨号参数, 类型指针
    dwNotifierType: DWORD; //消息通知方式
    lpvNotifier: DWORD; //消息处理事件
    var rasConn: HRASConn //返回成功连接的连接句柄
): DWORD; stdcall;
function RasEnumConnections(RASConn: PRasConn;
    var BufSize: DWord;
    var Connections: DWord
): LongInt; stdcall;
Function RasEnumEntries (
    reserved: PChar; //保留字段, 必须为空
    lpszPhonebook: PChar; //电话本名称, 在 Windows 9x 下无用, 可设置为 NIL
    lprasentryname: PRASENTRYNAME ; //接收拨号连接名称的缓冲区, 是数组的指针
    var lpcb: DWORD; //接收拨号连接名称的缓冲区的大小
    var lpcEntries: DWORD //实际获得拨号连接的数目
): DWORD; stdcall;
function RasGetConnectStatus(RASConn: hrasConn; //指定活动连接的句柄
    RASConnStatus: PRASConnStatus // 获取活动连接状态信息的缓冲区
): LongInt; stdcall;
function RasGetErrorString(ErrorCode: DWord; //错误代码标识
    szErrorString: PChar; //错误提示信息的缓冲区
    BufSize: DWord //错误提示信息的缓冲区的大小
): LongInt; stdcall;
function RasHangUp(RASConn: hrasConn): LongInt; stdcall;
function RasGetEntryDialParams(
    lpszPhonebook: PChar; //电话本名称, 在 Windows 9x 下无用, 可设置为 NIL
    VAR lprasdialparams: TRASDIALPARAMS; //拨号参数, 是一类型指针
    VAR lpfPassword: BOOL //显示是否需要用户密码

```

```

    ): DWORD; stdcall;
implementation
const
  RAS_DLL = 'RASAPI32';
function RasDial; external RAS_DLL name 'RasDialA';
function RasEnumConnections; external RAS_DLL name 'RasEnumConnectionsA';
function RasEnumEntries; external RAS_DLL name 'RasEnumEntriesA';
function RasGetConnectStatus; external RAS_DLL name 'RasGetConnectStatusA';
function RasGetErrorString; external RAS_DLL name 'RasGetErrorStringA';
function RasHangUp; external RAS_DLL name 'RasHangUpA';
function RasGetEntryDialParams; external RAS_DLL name 'RasGetEntryDialParamsA';
end.

```

9.3.5 使用拨号网络的类 Tras

从动态链接库中直接调用拨号网络的函数是一个很复杂且易出错的过程，通常这些直接从 DLL 中调用的函数都需要一长串结构复杂的、难以理解的参数，并且要实现一项具有实际意义的功能需要调用好几个这样的函数，因此一般都将这些函数用类的形式加以包装。

1. Tras 类

包装 RAS 的类称为“Tras”，该类定义的一些主要的属性与方法如下：

```

TRAS = class(TComponent)
private
  //私有声明
  FEntryName,      //拨号连接项
  FPhoneNumber,    //要连接的主机的电话号码
  FPhoneBookPath,  //电话簿路径
  FCallbackNumber, //回叫号码
  FUserName,       //用户名
  FPassword,       //用户密码
  FDomain,         //域
  FDeviceType,     //设备类型
  FClientIP,       //客户端地址
  FServerIP,       //服务器端地址
  FDeviceName,     //设备名称
  FDevicePort: String; //设备端口
  RASEvent: Word;  //RAS 事件
  RASLib: THandle; //RAS 库
  RASDialParams: TRASDialParams; //拨号参数
  RASAPI_Loaded: Boolean; //是否载入了动态链接库

```

```

... ..

procedure SetPhoneBookPath(Value: String);
procedure Connected;
procedure DisConnected;
function LoadRASAPI: boolean ;
procedure MoveDialParms ;
protected
  //Protected 声明
  RASConnect: Array[1..MaxConnections] OF TRASConn;
public
  //Public 声明
  PhoneBookEntries: TStringList;
  Connections: TConnectionList;

  Constructor Create(AOwner: TComponent); OVERRIDE;
  Destructor Destroy; override;
  function GetConnectStatus: LongInt;
  function Disconnect: LongInt;
  function GetErrorString(ErrorCode: LongInt): String;
  function Connect: LongInt;
  function CurrentStatus: String;
  function GetConnections: LongInt;
  function GetPhoneBookEntries: LongInt;
  function IntDisconnect: LongInt;
  function AutoConnect: LongInt;
  function LeaveOpen: LongInt;
  function ReOpen (item: integer) : LongInt;
  function GetDialParams: longInt;
  function SetDialParams: longInt;
  function EditPhonebook: LongInt ;
  function CreatePhonebook: LongInt ;
  function DeletePhonebook: LongInt ;
  function RenamePhonebook (newname: string): LongInt ;
  function ValidateName (newname: string): LongInt ;
  function GetIPAddress: LongInt;
  function GetEntryProperties: LongInt ;
  function SetEntryProperties: LongInt ;
  function GetConnection: String ;

```

Function

```
ChangePhoneNumber(NewEntryname:string;NewPhoneNumber:string):LongInt;
```

```
end;
```

Tras 类的具体定义和类中定义的函数、过程的实现，读者可在 Internet 上的有关 Delphi 的站点找到公共代码，在此由于篇幅所限，笔者不作详细讨论，下面只写出 Tras 类中定义的几个主要函数的实现代码，供读者参考。

//指定 Phonebook，进行拨号（用给定的 logon 和 password）

```
function TRAS.Connect: LongInt;
```

```
begin
```

```
  fLastError := ERROR_DLL_NOT_FOUND ;
```

```
  result := fLastError ;
```

```
  if NOT LoadRASAPI then
```

```
    exit ;
```

```
  if fRASConn <> 0 then //只允许一个连接
```

```
    IntDisconnect;
```

```
  fRASConn := 0;
```

```
  fConnectState := 0;
```

```
  fSavedState := 9999 ;
```

```
  ResetPerfStats ; // 清空 performance 统计
```

```
  MoveDialParams ;
```

```
  if fPhoneBookPath <> "" then
```

```
    fLastError := RasDial( Nil, PChar(fPhoneBookPath), @RASDialParams,
                           $FFFFFFFF, fWindowHandle, fRASConn)
```

```
  else
```

```
    fLastError:=RasDial( Nil, Nil, @RASDialParams, $FFFFFFFF,
                        fWindowHandle, fRASConn);
```

```
  if fLastError <> 0 then
```

```
    fStatusStr := GetErrorString (fLastError) ;
```

```
  Result := fLastError;
```

```
end;
```

//指定 Phonebook，得到 username/password/number，进行拨号

```
function TRAS.AutoConnect: LongInt;
```

```
begin
```

```
  GetDialParams ;
```

```
  if fLastError = 0 then
```

```
    GetEntryProperties ;
```

```
  if fLastError = 0 then
```

```

Connect;
result := fLastError;
end;

```

2. 在程序中实现拨号联网

下面以变电所遥测系统中的主站和子站之间通过拨号连接, 实现远程数据传输为例, 介绍如何在程序中通过使用 Tras 类实现拨号联网。

主站位于电力局, 拨号网络服务器端是主站的主机。子站是电力局各下属的变电所, 拨号网络客户端是子站的主机。系统中, 主站和子站都使用了拨号网络类 Tras (前文提到的 Tras 类); 主站是用于通过拨号给子站提供振铃信息, 然后立刻挂断。子站在监测到振铃信息后调用 Tras 拨号上网同主站建立网络连接以便进行 SOCKET 通信。下面分别介绍:

(1) 主站端调用 Tras

主站在需要和某变电所子站进行通信时, 先建立拨号网络。主站通过拨号再挂断的方法向子站提供振铃信号。实现过程如下:

先要在系统初始化的时候就创建 Tras 的实例 Ras:

```
Ras := Tras.create(self);
```

然后将需要通信的变电所的电话号码赋值给系统使用的拨号连接项 connect

注意: 在主站安装时需要建立一个拨号连接项取名为“connect”, 在需要连接其他变电所时, 只要更改其电话号码。

```
Ras.EntryName := 'connect';
```

```
Ras.ChangePhoneNumber('connect', PhoneNumber);
```

之后就可以拨号与挂断, 两者的间隔由该变电所参数表的字段 DELAY 来决定。挂断是在一临时线程中实现的。

```
CreateThread(nil, 0, @disconnect, nil, 0, ThreadID); //创建线程
```

```
ras.AutoConnect; // 拨号
```

先创建线程, 然后开始拨号。在该临时线程中调用函数 disconnect, 该函数执行完后线程结束。函数 Disconnect 的实现如下:

```

function disconnect:Longint; stdcall;
begin
    sleep(dial_delay);
    ras.DisConnect;
    result:=0;
end;

```

该函数在等待 dial_delay 毫秒后调用 ras.Disconnect 方法挂断电话。该延时的作用是使子站只有一或两声振铃。子站以此来判断是否是主站要求通信。

(2) 子站调用 TRas

子站在监测到振铃后, 通过调用拨号函数同主站建立网络连接。子站的振铃监测是在定时器中完成的, 在听到第一次振铃后, 开启另一定时器并开始对振铃次数记数。另一定时器定时间隔为 16 秒, 定时时间到后, 判断振铃为 1 或 2 次, 则表明主站要求通信, 并开始拨号,

否则认为是干扰，不作处理退出。

在子站安装时，也必须向安装主站一样先创建一个拨号连接项“connect”，该连接项的电话号码为主站的电话号码，用户名为主站为拨号上网所添加的用户名、密码。在程序中建立拨号连接的过程如下：

```
_closecomm;  
ras.EntryName:='connect';  
times:=0;  
how:=1;  
while (how<>0) and (times<2) do  
begin  
    ras.disconnect;  
    how:=ras.AutoConnect;  
    times:=times+1;  
end;
```

在该连接过程中，_closecomm 将 Modem 切换到自动应答方式。ras.disconnect 为连接挂断方法，ras.AutoConnect 为建立连接的方法，如果连接不成功则重试一次。

在连接不成功或通信完毕后，需要断开连接并切换 Modem 的应答方式，过程如下：

```
ras.disconnect;  
_closecomm;  
_opencomm('*****');
```

在该过程中，ras.disconnect 先断开拨号连接，然后串口，最后调用_opencomm 重新打开串口并切换到人工应答方式。

通过以上的过程，主站和子站之间就成功地建立了 RAS 连接，然后利用 Windows 的 Socket 机制，即可实现主站与子站之间的远程数据传输。

本章小结

本章介绍了 Windows 远程访问服务 (RAS) 的基本知识，Windows NT、Windows 2000 下拨号服务器的配置以及 Windows 95/98 下拨号客户端的配置。在本章的第三节，首先详细讨论了如何在程序中调用 RAS 函数实现拨号功能并给出具体实例，然后介绍如何使用 RAS 类实现 RAS 服务器与客户端之间的拨号联网，进行远程数据传输和几个 RAS 相关的 Delphi 实例。

第 10 章 通信安全设计

本章主要内容:

- 数据加密基础知识
- 应用编程接口编程模式和微软信息密码系统
- 创建签名消息和加密并封装一个消息
- 校验签名的消息和加密算法源码分析

在通信过程中,数据通过普通媒介进行传输,这导致数据很容易被他人截取。如果数据非常有价值,则窃密者截取后进行分析并加以利用,对于发送数据的人可能造成极大的损失。为了保护数据的安全,发送者需要采取一些措施,对数据加密就是一种可行的方法。

本章主要讲述了一些加密算法的基本概念,介绍了微软信息密码系统,利用 Microsoft Crypto Graphics 提供的函数,讲了几个数据加密处理的流程图,并给出了一个例子介绍如何访问 Microsoft CryptoAPI 提供的加密、解密服务。

10.1 数据加密基础知识

加密用来保护敏感信息的传输,保证信息的安全性。下面介绍一些基本加密技术、数字签名、数字信封的基本概念。

10.1.1 加密技术

密码学在过去发展了很久,总共有 3 项重要的技术牵扯到加密:不需要密钥的加密方式、需要一把密钥的加密方式以及需要两把密钥的加密方式。

目前一般采用两种加密体系:对称密钥加密和公开密钥加密。

1. 对称密钥加密

对称密钥加密(Symmetric Cryptography)这种方式中,加密和解密使用同一个密钥,私钥加密系统如图 10.1 所示。

数据加密标准 DES(Data Encryption Standard)是众所周知的对称密钥分组加密算法,该算法于 1995 年 3 月公开发表,并于 1997 年 1 月 15 日由美国国家标准局正式公布为数据加密标准,用于非国家保密机关,该算法已在银行和其他领域用了 20 年,它用于进行大批数据加密,加密时 DES 将数据分隔成 64bit 的数据块,并使用 56bit 的密钥对其进行一系列的数字变换。美国国家标准局曾就 DES 成立专门研究小组,研究结果认为,在 10~15 年内,DES

的安全性不成问题，在应用中将是能胜任的。

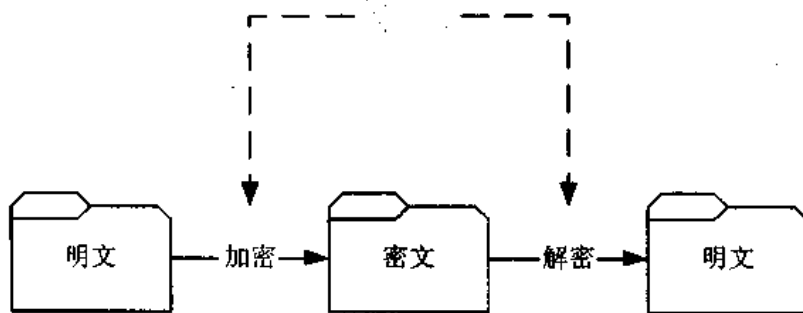


图 10.1 私钥加密系统

这种加密类型快速牢固，但能力有限，入侵者用一台运算能力足够强大的计算机依靠“野蛮力量”就能破译，RSA 实验室的研究人员使用一台价值 25 万美元的计算机在三天内就破解了 DES 算法。尽管如此，DES 作为乘积密码的典型代表在密码学发展历史上具有重要的地位。DES 的另一个不足是密钥的安全传送问题，密钥需单独进行交换以使接收者用以解密，因而，若密钥的传送不安全而被截获，则信息的加密形同虚设。

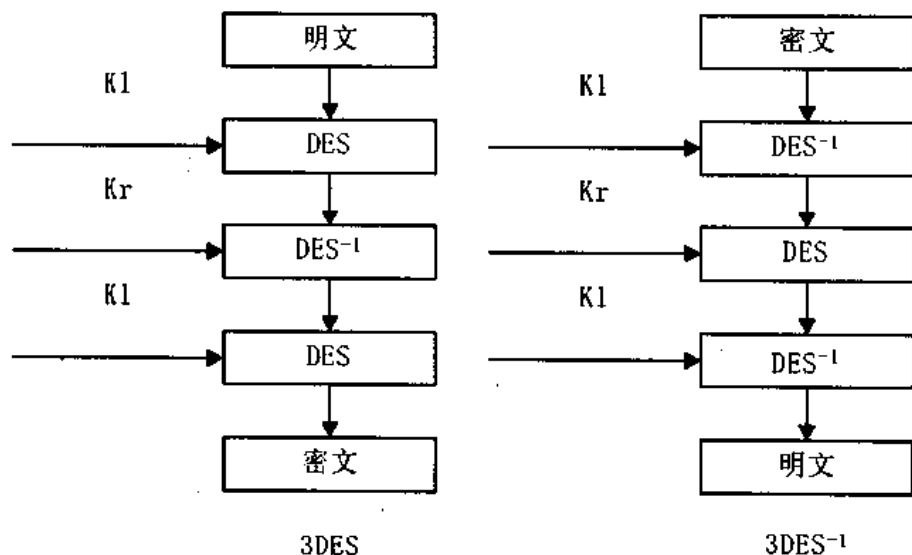


图 10.2 3DES 解密/解密流程图

下面介绍一个 DES 的算法改进版。

因为确定一种新的加密法是否真的安全是极为困难的，而且 DES 的惟一密码学缺点，就是密钥长度相对较短，所以人们并没有放弃使用 DES，而是想出了一个解决其长度问题的方法，即采用三重 DES。这种方法用两个密钥对明文进行三次加密，假设两个密钥是 K1 和 Kr，其算法的步骤如图 10.2 所示。

- (1) 用密钥 K1 进行 DES 加密。
- (2) 用 Kr 对步骤 1 的结果进行 DES 解密。
- (3) 用步骤 2 的结果使用密钥 K1 进行 DES 加密。

其中, DES 为数据加密标准算法, DES-1 为 DES 的反函数 (即解密函数)。3DES-1 为 3DES 的反函数。

这种方法的缺点, 是要花费原来三倍时间进行加密或解密处理, 从另一方面来看, 三重 DES 的 112 位密钥长度是很“强壮”的加密方式了。

2. 公开密钥加密

公开密钥加密是一种非对称密钥加密 (Asymmetric Cryptography), 它使用两个不同的密钥来加、解密, RSA 是著名的公开密钥加密算法, 如图 10.3 所示。

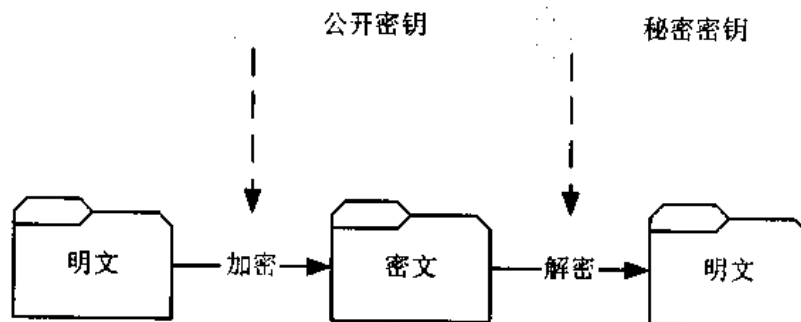


图 10.3 非对称密钥加密系统

RSA 算法是由 Rivest、Shamir 和 Adleman 三人研究发明的, 它的安全性是基于大数因子分解, 后者在数学上是一个难题, 至今没有有效的算法。

RSA 的原理是: 利用两个足够大的质数相乘所产生的乘积来加密和解密, 这两个质数无论哪一个先与原文编码相乘, 对原文加密, 均可由另一个质数再相乘来解密。这两个质数数学相关, 但要用其中一个求出另一个, 是十分困难的。

这一对质数称为密钥对 (Key Pair), 在应用时, 每个用户有一对, 并将其中一个公开, 称公开密钥 (Public Key), 发信者将信息用对方的公开密钥加密后给对方, 只有用该用户一个人知道的私有密钥 (Private Key) 才能解密。

公开密钥加密与对称密钥加密相比, 其优势在于不需要一把共享的通用密钥, 私钥只有用户自己知道。这样, 只有公钥而无与之匹配的私钥, 对入侵者没有任何用处。另一个用处是身份验证, 用私钥加密后意味着信息带有了加密者自身的特征, 接收者由此可知这条信息确实来自拥有私钥的一方。

凡具有 CA 证书的人的公钥可在网上查到, 亦可请对方发信息时传给你。

10.1.2 数字签名 (Digital Signature)

安全单向散列函数 (Secure Hash) 算法亦称安全哈希算法 (SHA: Secure Hash Algorithm) 或 MDS (MD Standards For Message Digest), 是由 Ron Rivest 所设计, 用于对要传输的数据作运算生成一串 128 位或 160 位的信息摘要, 这一摘要亦称为数字指纹 (Finger Print), 它的目的是为确保数据不被修改或变化, 保证信息的完整性不被破坏。

Hash 函数有 3 个主要特征。

□ 固定性和一致性: 它能处理任意大小的信息, 且生成固定大小的数据块, 对同一源

数据反复执行 Hash 函数将总是得到同样的结果。

❑ 不可预见性和敏感性：摘要的大小与原始信息的大小无任何联系，同时，源数据与产生的摘要看起来也无明显关系，源信息一个微小的变化都会对摘要产生很大的影响。

❑ 不可逆性：即没有办法通过产生的摘要直接恢复源数据。

签名是确认文件的一种手段，一般书面手工签名的作用有两种：一是由于自己的签名难以否认，从而确认了文件已签署这一事实；二是由于签名不易仿冒，从而确认了文件是真的这一事实。

数字签名的流程图如图 10.4 所示，采用数字签名也能确认以下两点：信息是由签名者发送的；信息自签发到收到为止，没做过任何修改。

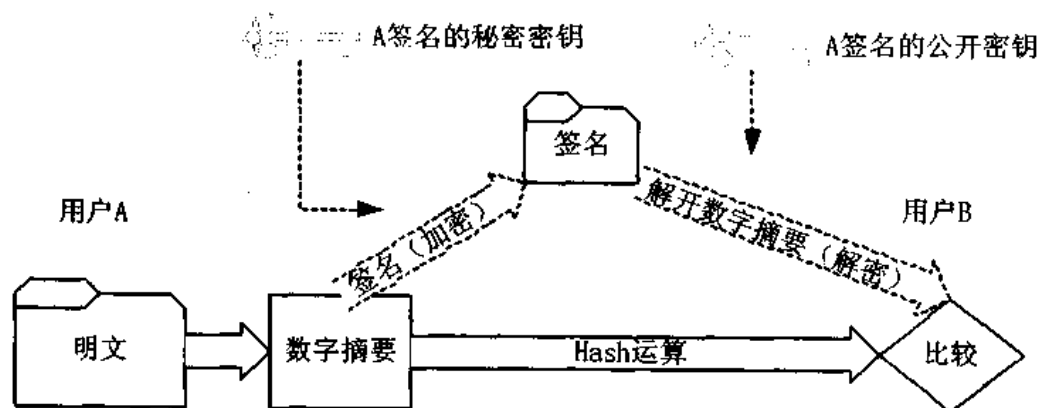


图 10.4 数字签名的流程图

Hash 函数与公开密钥算法是实现数字签名的主要技术，其原理是：发送者将信息用 Hash 函数对信息生成 128 位或 160 位的数字摘要；发送者用自己的私钥对摘要再加密，形成数字签名；发送者将信息本身和数字签名同时传给对方；接收者对收到的信息用 Hash 函数生成新的摘要，同时，用发送者的公开密钥对信息摘要解密；将解密后的摘要与新摘要对比，如两者一致，则说明传送过程中信息没有被破坏或篡改。

10.1.3 数字信封

数字信封技术结合了对称密钥和公开密钥加密技术上的优点，把对称密钥的快速、低成本和非对称密钥的有效性结合在一起，可克服对称密钥加密中密钥分发困难和公开密钥中加密时间长的问题。

数字信封如图 10.5 所示，其实现步骤如下：

发送方首先生成一个对称密钥，并用该密钥对要发送的信息加密得到密文 a；发送方用接收方的公开密钥加密上述对称密钥得到密文 b；将 a、b 传给接收方；接收方用自己的私钥解密被加密的对称密钥；接收方用得到的对称密钥对密文解密从而得到源报文。

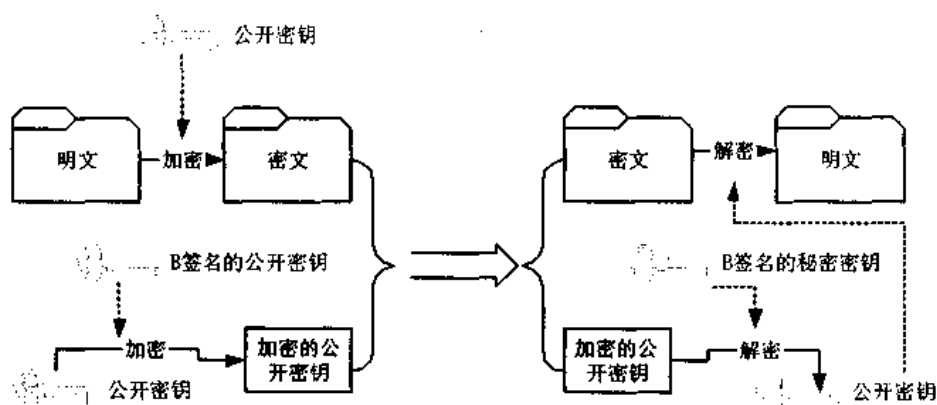


图 10.5 数字信封处理流程图

10.2 应用编程接口编程模式

密码应用编程接口（API）工作模式类似于 Windows 图形设备接口 GDI 工作模式。密码应用编程接口提供了一组独立于密码服务提供商（Cryptographic Service Provider）的标准密码指令集，在应用程序与密码服务提供商之间提供一个环境外壳，负责应用程序的密码服务要求与执行密码应用程序系统中的密码功能之间的转换。所有加密操作均由密码服务提供商来完成，包括加密数据或数字签名和用户私钥保护等。密码服务提供商与图形设备驱动程序类似，是可根据需要进行增删的独立模块。密码应用编程接口的外壳模式将应用程序分隔成了用户态与核心态。在用户态下，应用程序禁止直接与密码服务提供商通信，而是交给作为操作系统部分的密码应用编程接口来完成。

10.3 微软信息密码系统

微软信息密码系统结构由三部分组成，它们分别是密码应用程序、操作系统和密码服务提供商（CSP）。如图 10.6 所示。

图中密码服务提供商是真正完成密码工作的独立模块。理想的 CSP 应完全独立于具体应用程序，这样任何给定应用能运行在多种 CSP 上。一种 CSP 最少由动态连接库和签名文件组成。签名文件保证操作系统识别 CSP。操作系统定期认证签名确保 CSP 没被篡改过。而动态链接库中包含了一系列密码服务程序的具体实现。例如微软 RSA 基本服务提供商提供的动态库文件为 RSABASE.DLL 和一个签名文件，它是和 Windows 操作系统绑定在一起的。当安装新的 CSP 时，服务提供商的安装文件将自动修改系统的配置文件，同时该 CSP 的密码应用编程接口的配置信息将存储在注册表的本地机器位置。每一个密码服务提供商都负责管理自己的密钥库，该密钥库可作为 CSP 为每个用户创建的永久性密钥的仓库。每个密钥库可有一个或多个存储特定用户所有密钥对的密钥容器（Container）。

绝大多数的 CSP 通常为用户提供两组公钥/私钥对。一组密钥对又叫密钥交换对，通常用于加密会话密钥，这样会话密钥可以安全存放或与其他用户交换。另一组又叫数字签名密钥对，通常用于创建数字签名。在用户与服务提供商建立联系之前，用户必须有一个密钥库，该密钥库存储在注册表的当前用户位置。操作系统成功安装某种 CSP 并创建密钥库后，就可以与应用程序取得连接。应用程序通过 CSP 名称调用 CSP 联系指令。一旦连接成功，由操作系统将 CSP 装入内存，其密码服务功能将成为操作系统核心部分运行。

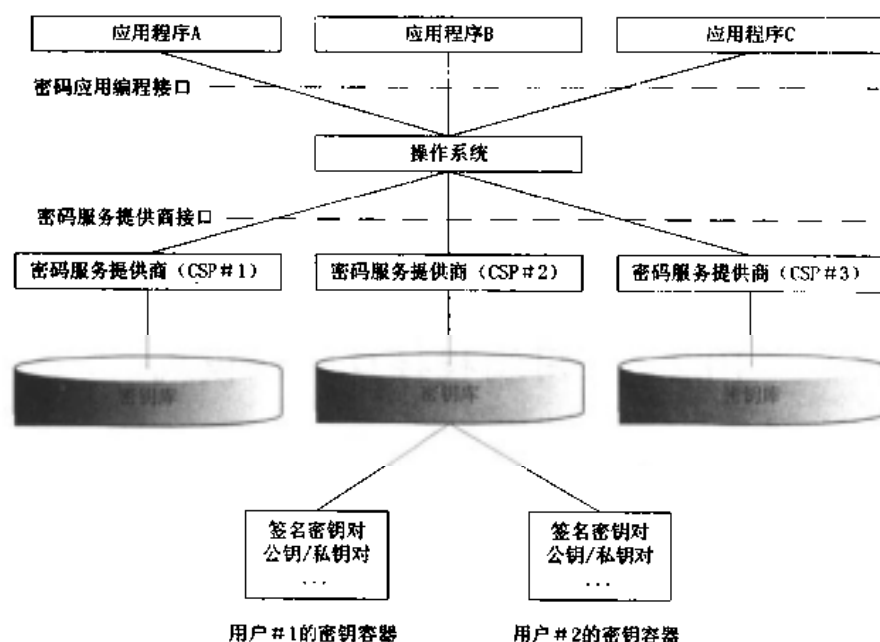


图 10.6 微软信息密码系统体系结构

表 10.1 列出了 Microsoft 公司提供的基本提供商 (Base Provider)，强提供商 (Strong Provider)，和增强提供商 (Enhanced Provider) 的差别。显示的密钥长度为缺省的密钥长度 (Key Length)。

用户可以使用 Microsoft® CryptoAPI (An Application Programming Interface) 提供的加密/解密服务，数字证书访问功能来实现数据的安全传输和保存。

下面章节介绍系统用到的关键加/解密处理。

表 10.1 Microsoft 公司提供的提供商比较

算法	基本提供商 密钥长度	强提供商密钥长度	增强提供商密钥长度
RSA 公钥签名算法	512 bits	512 bits	1 024 bits
RSA 公钥交换算法	512 bits	512 bits	1 024 bits
RC2 块加密算法	40 bits	40 bits	128 bits
RC4 流加密算法	40 bits	40 bits	128 bits
DES	56 bits	56 bits	56 bits
Triple DES (2 key)	不支持	112 bits	112 bits
Triple DES (3 key)	不支持	168 bits	168 bits

10.4 创建签名消息

流程图如图 10.7 所示。

(1) 获得指向欲签名数据的指针。

(2) 打开含有签名者证书的证书库。

(3) 获得对应该证书的私钥。在使用证书前，有两个属性必须设置，一个是用来将证书绑定到一个特定的 CSP (Cryptographies Service Provider)，在该 CSP 中，绑定到一个特定的私钥，另一个是用来指定当调用摘要操作时使用哪种哈希算法。这两个属性只需设置一次：从证书的属性中，决定哈希算法。

(4) 对数据使用哈希算法生成该数据的摘要。

(5) 使用通过证书属性获得的私钥，加密摘要，从而生成数据的签名；在签名的消息中包括如下信息：签名的数据 (Data To Be Signed)、哈希算法 (Hash Algorithm)、签名 (Digital Signature)、签名者的标识 (证书的颁发者和序列号) (Signer ID)、签名者的证书 (可选项) (My Certificate (Optional))。

下面给出关键函数的介绍。

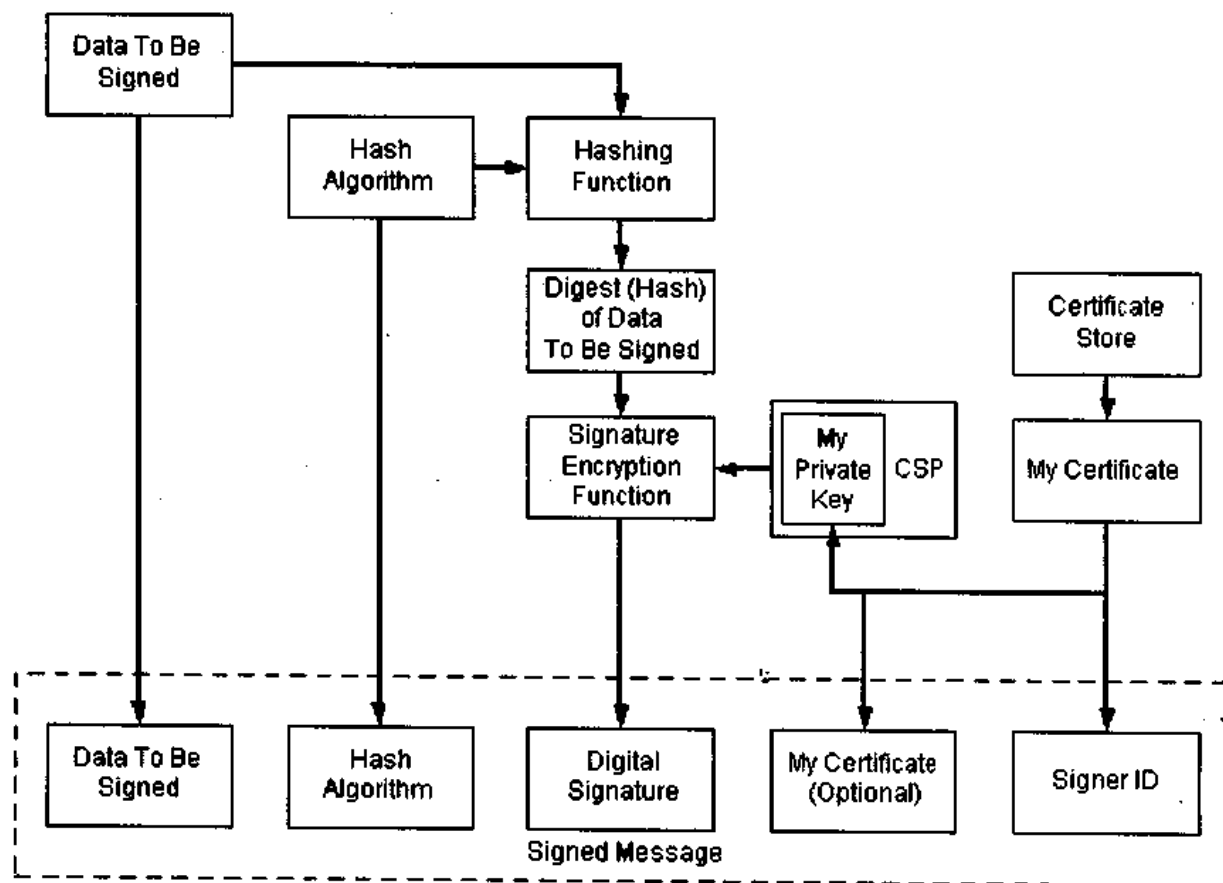


图 10.7 对消息签名

10.4.1 CertOpenStore

CertOpenStore 函数通过一个特定的库提供商类型打开一个证书库。

```
HCERTSTORE WINAPI CertOpenStore(
    LPCSTR lpszStoreProvider,
    DWORD dwMsgAndCertEncodingType,
    HCRYPTPROV hCryptProv,
    DWORD dwFlags,
    const void *pvPara
);
```

参数:

☐ lpszStoreProvider

指定库提供商类型 (Store Provider Type)。

☐ dwMsgAndCertEncodingType

仅适用于 CERT_STORE_PROV_MSG、CERT_STORE_PROV_PKCS7 或者 CERT_STORE_PROV_FILENAME 提供商类型。对于所有其他的提供商类型 (Provider Type)，此参数设置为 0。

☐ hCryptProv

加密提供商 (Cryptographic Provider) HCRYPTPROV 句柄。如果此参数设为 NULL，则函数使用一个适当的缺省的提供商。推荐使用缺省的提供商。缺省或者特定的加密提供商为所有校验一个目标证书 (Subject Certificate) 或者 CRL 的数字签名的库 API (Store API) 使用。

☐ dwFlags

该值包括高字 (High-Word) 和低字 (Low-Word)。dwFlags 的低字部分控制打开的证书库 (Certificate Store) 的各种各样的通用特征。

☐ pvPara

指向一个 VOID 的指针。

返回值:

如果函数执行成功，返回值为证书库的句柄；否则返回值为 NULL。

10.4.2 CertCloseStore

CertCloseStore 函数关闭证书库句柄 (Certificate Store Handle)，减少证书库的引用数 (Reference Count)。CertCloseStore 相应于 CertOpenStore 函数。

```
BOOL WINAPI CertCloseStore(
    HCERTSTORE hCertStore,
    DWORD dwFlags
);
```

参数:

☐ hCertStore

将被关闭的证书库的句柄。

☐ dwFlags

典型情况下，此参数使用缺省值 0。缺省值用于关闭库，内存仍然保留着分配给未释放的上下文。

返回值：

如果函数成功执行，则返回值为非零值（TRUE）。如果函数执行失败，则返回值为 0（FALSE）。要取得扩展的错误信息，请调用 GetLastError 函数。

10.4.3 CryptSignMessage

CryptSignMessage 函数生成一个特定内容的哈希值，签名哈希值，然后对原始消息内容和签名的哈希值编码。

```
BOOL WINAPI CryptSignMessage(
    PCRYPT_SIGN_MESSAGE_PARA pSignPara,
    BOOL fDetachedSignature,
    DWORD cToBeSigned,
    const BYTE *rgpbToBeSigned[,
    DWORD rgcbToBeSigned[,
    BYTE *pbSignedBlob,
    DWORD *pcbSignedBlob
);
```

参数：

☐ pSignPara

指向含有签名参数的 CRYPT_SIGN_MESSAGE_PARA 结构的指针。

☐ fDetachedSignature

如果是分离的签名则为 TRUE，否则，为 FALSE。如果为 TRUE，则仅签名的哈希值（Signed Hash）被编码到 pbSignedBlob，否则，rgpbToBeSigned 和签名的哈希值均被编码。

☐ cToBeSigned

在 rgpbToBeSigned 和 rgcbToBeSigned 中数组元素数目（The Number Of Array Elements）的计数。除非 fDetachedSignature 设为 TRUE，否则此参数必须设为 1。

☐ rgpbToBeSigned

指向含有需要签名的内容的缓冲区的指针的数组。

☐ rgcbToBeSigned

内容的大小（以字节为单位）数组。

☐ pbSignedBlob

指向接收编码签名哈希（Encoded Signed Hash）的缓冲区的指针，如果 fDetachedSignature 为 TRUE；指向接收编码内容和签名哈希（Encoded Signed Hash）的缓冲区的指针。

☐ pcbSignedBlob

当函数返回时，此变量含有已签名和编码的消息的大小（以字节为单位）。

返回值：

如果函数成功执行，则返回值为非零值 (TRUE)。如果函数执行失败，则返回值为 0 (FALSE)。要取得扩展的错误信息，请调用 GetLastError 函数。

10.5 加密并封装一个消息

加密并封装一个消息的流程图在图 10.8 中给出。

消息 (Message) 签完名之后, 需要对生成的签名的消息 (Signed Message) 进行封装处理, 然后才能使用 E-mail 发送出去。

数据封装图如图 10.8 所示, 下面给出处理顺序。

- (1) 取得指向欲封装数据的指针。
- (2) 生成一个对称（会话）密钥。
- (3) 使用对称密钥和指定的加密算法加密数据。
- (4) 打开证书库。
- (5) 从库中取出接收者的证书。
- (6) 从接收者的证书中取出公钥。
- (7) 使用接收者的公钥加密对称（会话）密钥。

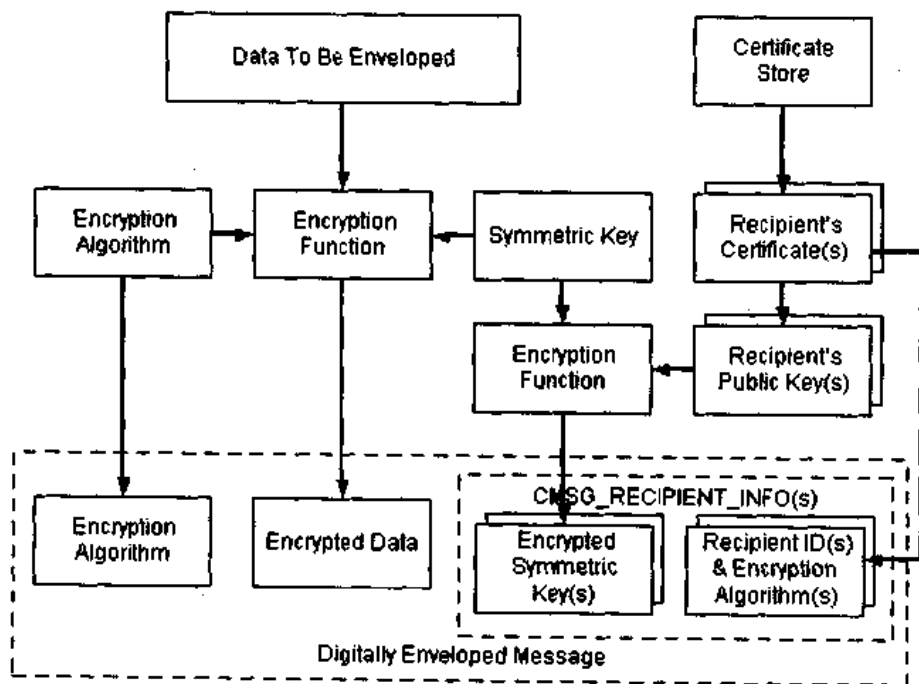


图 10.8 加密并封装一个消息

- (8) 从接收者的证书中, 取出接收者的 ID。
- (9) 数字封装消息中包括下列信息。

- ❑ 数据加密算法 (Data Encryption Algorithm)
- ❑ 加密的数据 (Encrypted Data)
- ❑ 加密的对称密钥 (Encrypted Symmetric Key)

- ❑ 接收者的标识符 (Recipient Identifier)

下面开始介绍关键函数。

10.5.1 CryptMsgOpenToEncode

CryptMsgOpenToEncode 函数打开一个密文消息用于编码并返回一个打开消息的句柄。消息一直打开，直到调用 CryptMsgClose 函数。

```
HCRYPTMSG WINAPI CryptMsgOpenToEncode(
    DWORD dwMsgEncodingType,
    DWORD dwFlags,
    DWORD dwMsgType,
    const void *pvMsgEncodeInfo,
    LPSTR pszInnerContentObjID,
    PCMSG_STREAM_INFO pStreamInfo
);
```

参数:

- ❑ dwMsgEncodingType

指定一个编码类型 (Encoding Type)。

- ❑ dwFlags

指定函数处理方式。

- ❑ dwMsgType

指定消息类型。

- ❑ pvMsgEncodeInfo

指向被编码数据的指针。指向的数据类型取决于 dwMsgType 的值。

- ❑ pszInnerContentObjID

如果调用 CryptMsgCalculateEncodedLength 并且 CryptMsgUpdate 的数据已经被消息编码，则适当的对象标识被传入 pszInnerContentObjID 中。如果 pszInnerContentObjID 为 NULL，则内部内容类型被认为以前没有编码过，编码为八位字符串 (Octet String) 并给定类型 CMSG_DATA。

- ❑ pszInnerContentObjID

如果调用 CryptMsgCalculateEncodedLength 并且 CryptMsgUpdate 的数据已经被消息编码，则适当的对象标识被传入 pszInnerContentObjID 中。如果 pszInnerContentObjID 为 NULL，则内部内容类型被认为以前没有编码过，编码为八位字符串 (Octet String) 并给定类型 CMSG_DATA。

注意：当使用流时，pszInnerContentObjID 必须为 NULL 或者 szOID_RSA_data。

- ❑ pStreamInfo。

当不使用流时，此参数设为 NULL。当使用流时，此参数指向一个 CMSG_STREAM_INFO 结构。

返回值:

函数成功执行，返回一个打开消息的句柄。如果失败，返回 NULL。

10.5.2 CryptMsgUpdate

CryptMsgUpdate 函数增加欲处理的内容到一个密文中。通过重复调用 CryptMsgUpdate 函数可以一段一段地处理消息。根据消息决定是用 CryptMsgOpenToEncode 函数还是用 CryptMsgOpenToDecode 函数增加的消息被编码或者解码。

```
BOOL WINAPI CryptMsgUpdate(
    HCRYPTMSG hCryptMsg,
    const BYTE *pbData,
    DWORD cbData,
    BOOL fFinal
);
```

参数:

☐ hCryptMsg

欲更新的消息的密文消息句柄。

☐ pbData

指向欲编码或解码的数据的缓冲区的指针。

☐ cbData

在 pbData 缓冲区中的数据量（以字节为单位）。

☐ fFinal

指定是否处理的编码或解码的数据块是最后一块。正确地使用该标志取决于被处理消息有附加数据。

返回值:

如果函数成功执行，则返回值为非零值（TRUE）。如果函数执行失败，则返回值为 0（FALSE）。要取得扩展的错误信息，请调用 GetLastError。

10.5.3 CryptMsgGetParam

在一个密文消息被编码或者解密后，CryptMsgGetParam 函数获得一个消息参数（Message Parameter）。在此 CryptMsgUpdate 最后一次调用后，此函数调用。

```
BOOL WINAPI CryptMsgGetParam(
    HCRYPTMSG hCryptMsg,
    DWORD dwParamType,
    DWORD dwIndex,
    void *pvData,
    DWORD *pcbData
);
```

参数:

☐ hCryptMsg

一个密文消息（Cryptographic Message）的句柄。

☐ dwParamType

指定被检索的数据的参数类型。被检索的数据的类型决定 pvData 使用的结构类型。

☐ dwIndex

当可用时，为被检索的参数的索引。当不可用时，此参数被忽略，并设为 0。

☐ pvData

指向接收被检索数据的缓冲区的指针。数据的形式是多样的，取决于参数类型。

☐ pcbData

指向一个变量的指针，此变量指定 pvData 参数指定的缓冲区的大小（以字节为单位）。

当函数返回时，此变量含有存储在缓冲区中的字节的数量。

返回值：

如果函数成功执行，则返回值为非零值（TRUE）。如果函数执行失败，则返回值为 0（FALSE）。要取得扩展的错误信息，请调用 GetLastError 函数。

10.5.4 CryptMsgClose

CryptMsgClose 函数关闭一个密文消息句柄（Cryptographic Message Handle）。在每次调用此函数时，对消息的引用数（Reference Count）减一。当引用数为 0 时，消息彻底释放。

BOOL WINAPI CryptMsgClose(

HCRYPTMSG hCryptMsg

);

参数：

☐ hCryptMsg

欲关闭的密文消息的句柄。

返回值：

如果函数成功执行，则返回值为非零值（TRUE）。如果函数执行失败，则返回值为 0（FALSE）。要取得扩展的错误信息，请调用 GetLastError。

10.6 解密封装的数据（或者解封数据）

用户接收到封装的数据之后，剥离需要解封的数据，首先对该数据解封，然后，校验消息是否完整无误。

数据解封图如图 10.9 所示，下面给出处理顺序：

- （1）取得指向数字封装消息（欲解封数据）的指针。
- （2）打开证书库。
- （3）从消息中取得接收者的 ID（My ID）。
- （4）根据接收者的 ID 获取证书。
- （5）取得与该证书相关联的私钥。
- （6）用私钥解开对称（会话）密钥。
- （7）从消息中取得加密算法。

(8) 使用私钥和加密算法解密数据。

如果数据被正确地解封, 接下来接收端进行消息校验处理; 否则, 表示数据包不对, E-mail 中途可能被篡改, 接收端向发送者发出 E-mail 接收出错通知。

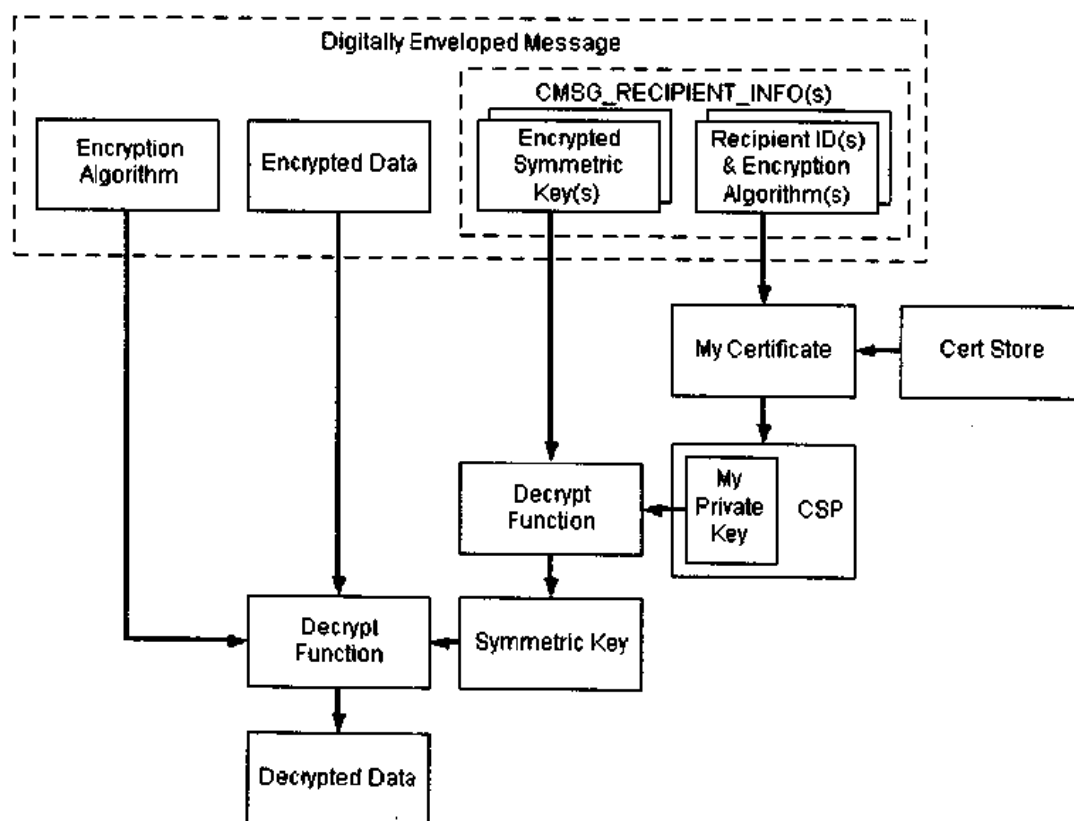


图 10.9 解封数据的流程图

下面给出关键函数的介绍。

CryptMsgOpenToDecode

CryptMsgOpenToDecode 函数打开一个密文消息用于解码并返回一个打开消息的句柄。消息一直打开, 直到调用 CryptMsgClose。

```

HCRYPTMSG WINAPI CryptMsgOpenToDecode(
    DWORD dwMsgEncodingType,
    DWORD dwFlags,
    DWORD dwMsgType,
    HCRYPTPROV hCryptProv,
    PCERT_INFO pRecipientInfo,
    PCMSG_STREAM_INFO pStreamInfo
);
  
```

参数:

□ dwMsgEncodingType

指定一个编码类型 (Encoding Type)。

❑ dwFlags

指定函数处理方式。

❑ dwMsgType

大多数情况下, 消息类型 (Message Type) 由消息头 (Message Header) 决定并且此参数设为 0。如果没有消息头并且此参数设为 0, 则此函数执行失败。

❑ hCryptProv

为密码提供商用于哈希消息指定一个句柄。签名消息, 则 hCryptProv 用于签名校验。

❑ pRecipientInfo

此参数为未来使用而保留, 必须设为 NULL。

❑ pStreamInfo

当不使用流时, 此参数必须设为 NULL。

返回值:

函数成功执行, 返回一个打开消息的句柄。如果失败, 返回 NULL。

10.7 校验签名的消息

校验签名的消息的流程图如图 10.10 所示。

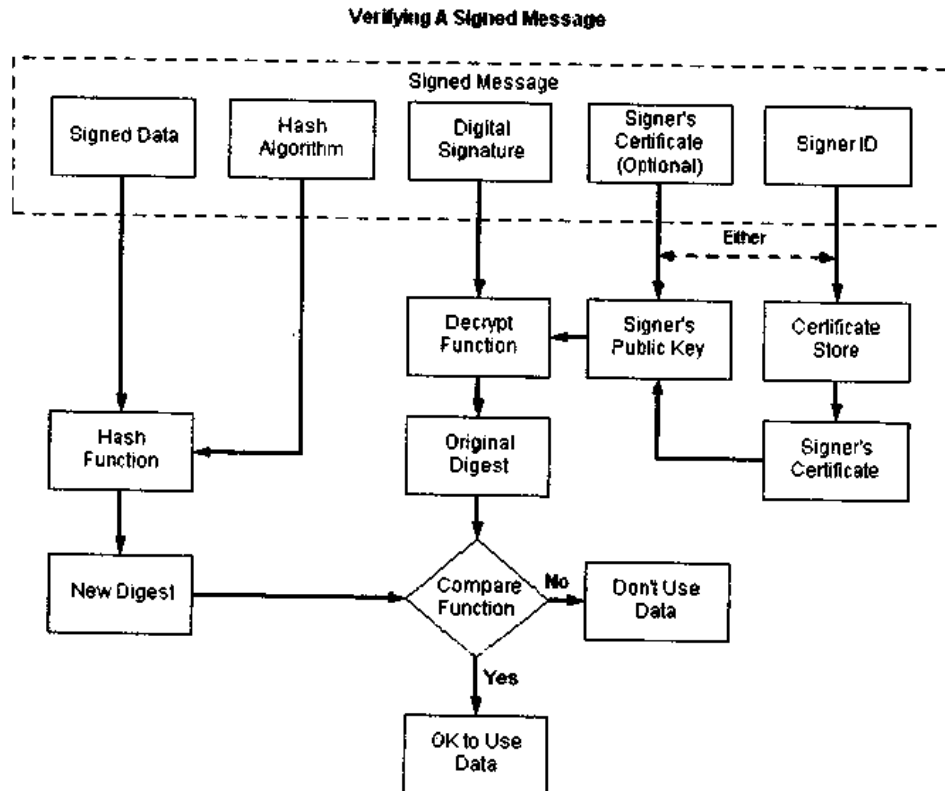


图 10.10 校验签名的消息的流程图

校验签名消息 (Signed Message) 的签名:

(1) 获得指向签名消息的指针。

(2) 打开证书库。

(3) 使用消息中签名者的 ID, 获得发送者的证书和其上的公钥的句柄; 作为步骤 (2) 和 (3) 的可选方案, 可以使用包含在消息中的证书来取得签名者的公钥。

(4) 使用签名者的公钥, 解密数字签名, 生成在消息中的数据原始摘要。

(5) 获得包含在消息中的哈希算法, 对包含在消息中的数据运用该哈希算法, 生成一个新的摘要。

(6) 比较从消息中取得的摘要和刚刚生成的新摘要。

(7) 如果两个摘要匹配, 签名校验正确。这意味着用来对数据签名的私钥和刚才用来解密签名的公钥是匹配的, 并且自数据签名之后, 数据未被更改过; 如果两个摘要不匹配, 签名校验不通过, 并且或者私钥和公钥不匹配, 或者自数据签名之后, 数据被更改过; 或者两者均是; 根据需要, 接收端将检验结果通过 E-mail 通知发送方正确或者未能正确接收信件。

关键函数 CryptVerifyMessageSignature

CryptVerifyMessageSignature 函数校验一个签名的消息的签名 (a signed message's signature)。

```
BOOL WINAPI CryptVerifyMessageSignature(
    PCRYPT_VERIFY_MESSAGE_PARA pVerifyPara,
    DWORD dwSignerIndex,
    const BYTE *pbSignedBlob,
    DWORD cbSignedBlob,
    BYTE *pbDecoded,
    DWORD *pcbDecoded,
    PCCERT_CONTEXT *ppSignerCert
);
```

参数:

□ pVerifyPara

指向一个包含校验参数 (Verification Parameters) 的 CRYPT_VERIFY_MESSAGE_PARA 结构的指针。

□ dwSignerIndex

想要的签名的索引。能够有多个签名。CryptVerifyMessageSignature 能被重复调用, 每次增加 dwSignerIndex。对于第一个签名者, 设定此参数为 0。如果函数返回值为 FALSE, 并且 GetLastError 返回 CRYPT_E_NO_SIGNER, 则前一次调用处理的是消息中的最后一个签名者。

□ pbSignedBlob

指向含有签名的消息的缓冲区的指针。

□ cbSignedBlob

签名的消息的缓冲区的大小 (以字节为单位)。

□ pbDecoded

指向接收解码消息的缓冲区的指针。

□ pcbDecoded

指向指定 pcbDecoded 缓冲区的大小（以字节为单位）的一个 DWORD 的指针。当函数返回时，这个 DWORD 含有解码消息的大小（以字节为单位）。如果此参数设为 NULL，则不会返回解码消息。

□ ppSignerCert

指向含有签名者证书的 CERT_CONTEXT 的指针。如果签名者的证书不需要，则此参数设为 NULL。

返回值：

如果函数成功执行，则返回值为非零值（TRUE）。如果函数执行失败，则返回值为 0（FALSE）。要取得扩展的错误信息，请调用 GetLastError。

10.8 加密算法源码分析

DLL 独立于编程语言，大多数 Windows 编程环境都允许主程序调用 DLL 中的函数。

可以用 Visual C++、Visual Basic、Power Builder、Delphi、汇编语言等建立 DLL，然后在不同语言编制的应用程序中调用它。这里用 Visual C++ 编译 DLL，在 Delphi 中调用了它。

本节应用了动态连接库（DLL）方法，详细用法请参见 MSDN 和 Delphi 的帮助文档。

下面给出的程序是使用了 Microsoft CryptoAPI 提供的加密、解密服务对数据进行处理。程序编译后生成一个 DLL 文件，其中封装了两个加密、解密的函数。读者可以在 Delphi 中调用其中的函数。在源码分析完后，将给出一个例子，介绍如何在 Delphi 中调用这两个函数。

证书列表为笔者的证书，如图 10.11 所示。

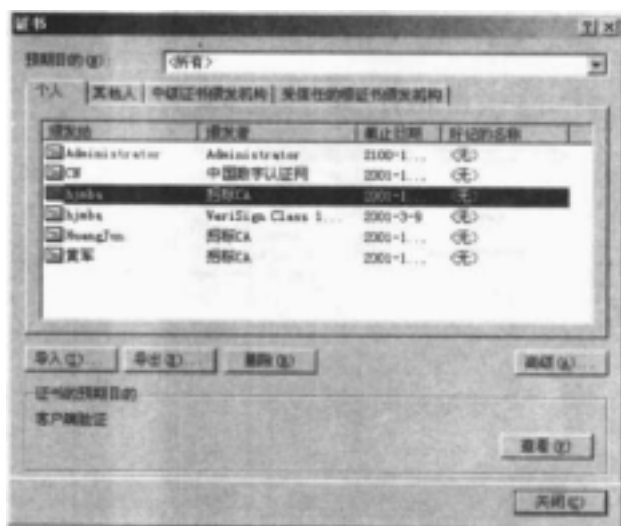


图 10.11 证书列表

本 DLL 文件的运行平台建议为 Windows 2000 Server 或者 Advance Windows 2000 Server。笔者在 Windows 2000 Server 下，用 Visual Studio 6.0 编译该 DLL 项目。其中的 Delphi 例子使

用 Delphi 5.0, 在 Windows 2000 Server 下编译成功。

其中使用的函数的具体说明, 请参见 MSDN Library Visual Studio 6.0 或者最新的 MSDN。

注意: 当调用 DLL 库中的解密或者解密函数时, 如果传入的参数中密码为空, 解密或者解密函数将使用缺省的密码容器。为了使用加密者的用于密钥交互的密钥, 用户使用的计算机上应该安装个人的密码容器。笔者的计算机上安装了个人数字证书和个人的密码容器。读者在自己的计算机上申请一个数字证书, 就能使用该功能了。

10.8.1 加密、解密函数库

项目的创建步骤如下:

(1) 启动 Visual Studio。

(2) 创建一个新工程, 名字为 EncryptService, 项目类型为 Win32 Dynamic-Link Library, 单击 OK 按钮, 如图 10.12 所示。

(3) 选择创建一个空的 DLL 工程, 单击 Finish 按钮, 如图 10.13 所示。

(4) 单击 OK 按钮, Visual Studio 自动创建一个工程。

(5) 向项目中添加 main.h 和 main.cpp 两个文件。

(6) 编译。

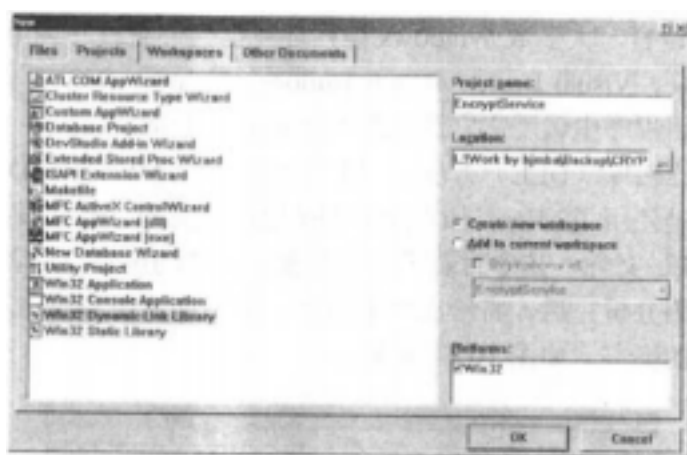


图 10.12 项目创建步骤 1

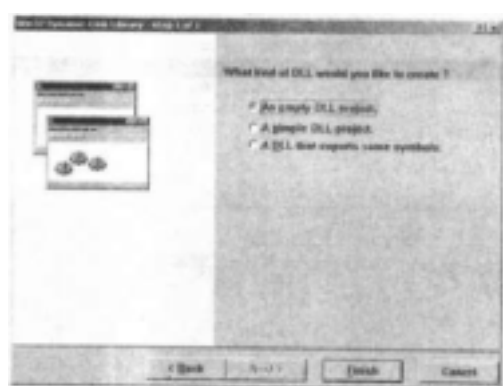


图 10.13 项目创建步骤 2

1. 头文件 main.h

头文件 (header file) main.h, 提供函数声明。

```
#include "stdio.h"
#include "windows.h"
#include "wincrypt.h" //加密/解密服务相关的声明

#define ENCRYPTDLL_API __declspec(dllexport) //函数输出标志

#define MY_ENCODING_TYPE (PKCS_7_ASN_ENCODING | X509_ASN_ENCODING)
//-----
//要求声明#define
#define ENCRYPT_ALGORITHM CALG_RC2    可用
#define ENCRYPT_ALGORITHM CALG_DES    可用

#define ENCRYPT_ALGORITHM CALG_3DES
#define ENCRYPT_BLOCK_SIZE 8

extern "C" extern ENCRYPTDLL_API int fnEncryptDLL(void);

//-----
// 加密函数
// 参数说明:
//     szSource: 欲加密的文件, 原文件或者明文
//     szDestination: 已经加密的文件, 目标文件
//     szPassword: 加密用的密码, 如果不需要密码, 则赋值为 NULL
//               如果需要密码, 则赋值密码串
//-----

extern "C" extern ENCRYPTDLL_API BOOL MyEncryptFile(
    PCHAR szSource,
    PCHAR szDestination,
    PCHAR szPassword);

//输出出错信息。
extern "C" extern ENCRYPTDLL_API void HandleError(char *s);

//-----
// 解密函数
// 参数说明:
```

```
//      szSource: 欲解密的文件, 原文件
//      szDestination: 已经解密的文件, 目标文件
//      szPassword: 解密用的密码, 如果不需要密码, 则赋值为 NULL
//                      如果需要密码, 则赋值密码串
//-----
```

```
extern "C" extern ENCRYPTDLL_API BOOL MyDecryptFile(
    PCHAR szSource,
    PCHAR szDestination,
    PCHAR szPassword);
```

2. 主体文件 main.cpp

下面的文件为主体文件 main.cpp。

```
#include "Main.h"
```

```
//-----
```

```
// 如下是一个输出函数的例子
```

```
//-----
```

```
extern "C" extern ENCRYPTDLL_API int fnEncryptDLL(void)
{
    return 42;
}
```

```
extern "C" extern ENCRYPTDLL_API BOOL MyEncryptFile(
    PCHAR szSource,
    PCHAR szDestination,
    PCHAR szPassword)
```

```
//-----
```

```
// 参数说明:
```

```
//      szSource: 欲加密的文件, 原文件或者明文
```

```
//      szDestination: 已经加密的文件, 目标文件
```

```
//      szPassword: 加密用的密码, 如果不需要密码, 则赋值为 NULL
```

```
//                      如果需要密码, 则赋值密码串
```

```
//-----
```

```
{
    // 声明并初始化局部变量
```

```
    FILE *hSource;
```

```
    FILE *hDestination;
```

```

HCRYPTPROV hCryptProv;
HCRYPTKEY hKey;
HCRYPTKEY hXchgKey;
HCRYPTHASH hHash;

PBYTE pbKeyBlob;
DWORD dwKeyBlobLen;

PBYTE pbBuffer;
DWORD dwBlockLen;
DWORD dwBufferLen;
DWORD dwCount;

//-----
// 打开源文件
if(hSource = fopen(szSource,"rb"))
{
    printf("The source plaintext file, %s, is open. \n", szSource);
}
else
{
    HandleError("Error opening source plaintext file!");
}

//-----
// 打开目标文件
if(hDestination = fopen(szDestination,"wb"))
{
    printf("Destination file %s is open. \n", szDestination);
}
else
{
    HandleError("Error opening destination ciphertext file!");
}
// 获取指向缺省提供商的句柄
// 使用笔者个人的密码容器
// 读者在自己的机器上运行本函数时, 请
// 使用自己的密码容器名

```

```
if(CryptAcquireContext(
    &hCryptProv,
    "黄军",          //NULL
    NULL,
    PROV_RSA_FULL,
    0))
{
    printf("A cryptographic provider has been acquired. \n");
}
else
{
    HandleError("Error during CryptAcquireContext!");
}
//-----
// 创建会话密钥

if(!szPassword )
{
    //-----
    // 没有传入密码
    // 用随机会话密钥 (random session key) 加密文件
    // 并将密钥写入文件
    // 创建一个随机会话密钥 (random session key)

    if(CryptGenKey(
        hCryptProv,
        ENCRYPT_ALGORITHM,
        CRYPT_EXPORTABLE,
        &hKey))
    {
        printf("A session key has been created. \n");
    }
    else
    {
        HandleError("Error during CryptGenKey. \n");
    }
    //-----
    // 获取指向加密者交换公钥的句柄
```

```

if(CryptGetUserKey(
    hCryptProv,
    AT_KEYEXCHANGE,
    &hXchgKey))
{
    printf("The user public key has been retrieved. \n");
}
else
{
    HandleError("User public key is not available \
        and may not exist.");
}
//-----
// 确定密钥块 (key blob) 的大小, 并分配内存

if(CryptExportKey(
    hKey,
    hXchgKey,
    SIMPLEBLOB,
    0,
    NULL,
    &dwKeyBlobLen))
{
    printf("The key blob is %d bytes long. \n",dwKeyBlobLen);
}
else
{
    HandleError("Error computing blob length! \n");
}
if(pbKeyBlob =(BYTE *)malloc(dwKeyBlobLen))
{
    printf("Memory is allocated for the key blob. \n");
}
else
{
    HandleError("Out of memory. \n");
}
//-----
// 加密并输出会话密钥到一个简单密钥块 (simple key blob)

```

```
if(CryptExportKey(
    hKey,
    hXchgKey,
    SIMPLEBLOB,
    0,
    pbKeyBlob,
    &dwKeyBlobLen))
{
    printf("The key has been exported. \n");
}
else
{
    HandleError("Error during CryptExportKey!\n");
}

//-----
// 释放用于密钥交换的密钥句柄
CryptDestroyKey(hXchgKey);
hXchgKey = 0;

//-----
// 将密钥块大小写入到目标文件
fwrite(&dwKeyBlobLen, sizeof(DWORD), 1, hDestination);
if(ferror(hDestination))
{
    HandleError("Error writing header.");
}
else
{
    printf("A file header has been written. \n");
}

//-----
// 将密钥块写入到目标文件

fwrite(pbKeyBlob, 1, dwKeyBlobLen, hDestination);
if(ferror(hDestination))
{
    HandleError("Error writing header");
```

```

    }
    else
    {
        printf("The key blob has been written to the file. \n");
    }
}
else
{
    //-----
    // 文件用从密码 (password) 派生的会话密钥加密
    // 仅当用于创建会话密钥的密码存在, 文件解密时
    // 会话密钥才重新创建

    //-----
    // 创建一个哈希对象 (hash object)

    if(CryptCreateHash(
        hCryptProv,
        CALG_MD5,
        0,
        0,
        &hHash))
    {
        printf("A hash object has been created. \n");
    }
    else
    {
        HandleError("Error during CryptCreateHash!\n");
    }

    //-----
    // 对密码进行哈希运算

    if(CryptHashData(
        hHash,
        (BYTE *)szPassword,
        strlen(szPassword),
        0))
    {

```



```

        printf("The password has been added to the hash. \n");
    }
    else
    {
        HandleError("Error during CryptHashData. \n");
    }
//-----
// 从哈希对象 (hash object) 派生 (derive) 出会话密钥 (session key)

    if(CryptDeriveKey(
        hCryptProv,
        ENCRYPT_ALGORITHM,
        hHash,
        0,
        &hKey))
    {
        printf("An encryption key is derived from the password hash. \n");
    }
    else
    {
        HandleError("Error during CryptDeriveKey!\n");
    }
//-----
// 注销 (destroy) 哈希对象

    CryptDestroyHash(hHash);
    hHash = 0;
}
//-----
// 会话密钥现在准备好了。如果它不是派生于一个密码的密钥
// 用加密者的私钥 (encrypter's private key) 加密的会话
// 密钥 (session key) 将被写到目标文件 (destination file)

//-----
// 确定一次加密的字节数
// 其值必需是 ENCRYPT_BLOCK_SIZE 的倍数
// ENCRYPT_BLOCK_SIZE 在#define 声明中给出

    dwBlockLen = 1000 - 1000 % ENCRYPT_BLOCK_SIZE;

```

```

//-----
// 确定块大小。如果使用块密码 (block cipher), 则必需有
// 附加块的空间 (room for an extra block)

if(ENCRYPT_BLOCK_SIZE > 1)
    dwBufferLen = dwBlockLen + ENCRYPT_BLOCK_SIZE;
else
    dwBufferLen = dwBlockLen;

//-----
// 分配内存
if(pbBuffer = (BYTE *)malloc(dwBufferLen))
{
    printf("Memory has been allocated for the buffer. \n");
}
else
{
    HandleError("Out of memory. \n");
}
//-----
// 在一个 do 循环中, 加密源文件并写到目标文件中
do
{
//-----
// 从源文件中读 dwBlockLen 字节
dwCount = fread(pbBuffer, 1, dwBlockLen, hSource);
if(ferror(hSource))
{
    HandleError("Error reading plaintext!\n");
}

//-----
// 加密数据
if(!CryptEncrypt(
    hKey,
    0,
    feof(hSource),
    0,
    pbBuffer,

```

```
        &dwCount,
        dwBufferLen))
    {
        HandleError("Error during CryptEncrypt. \n");
    }

//-----
// 写数据到目标文件
fwrite(pbBuffer, 1, dwCount, hDestination);
if(ferror(hDestination))
{
    HandleError("Error writing ciphertext.");
}

}
while(!feof(hSource));
//-----
// 当源文件最后一块被读出，加密并写到目标文件时
// 结束 do 循环

//-----
// 关闭文件

if(hSource)
    fclose(hSource);
if(hDestination)
    fclose(hDestination);

//-----
// 释放内存

if(pbBuffer)
    free(pbBuffer);

//-----
// 注销会话密钥 (Destroy session key)

if(hKey)
    CryptDestroyKey(hKey);
```

```

//-----
// 释放密钥交换密钥句柄

if(hXchgKey)
    CryptDestroyKey(hXchgKey);

//-----
// 注销哈希对象

if(hHash)
    CryptDestroyHash(hHash);

//-----
// 释放提供商句柄

if(hCryptProv)
    CryptReleaseContext(hCryptProv, 0);
return(TRUE);
} // Encryptfile 结束

extern "C" extern ENCRYPTDLL_API void HandleError(char *s)
{
    // 错误处理
/*  fprintf(stderr, "An error occurred in running the program. \n");
    fprintf(stderr, "%s\n", s);
    fprintf(stderr, "Error number %x.\n", GetLastError());
    fprintf(stderr, "Program terminating. \n");
    exit(1);
*/
} // HandleError 结束

//-----
// 定义 Decryptfile 函数

extern "C" extern ENCRYPTDLL_API BOOL MyDecryptFile(
    PCHAR szSource,
    PCHAR szDestination,
    PCHAR szPassword)
//-----

```

```
// 解密函数
// 参数说明
//     szSource: 欲解密的文件, 原文件
//     szDestination: 已经解密的文件, 目标文件
//     szPassword: 解密用的密码, 如果不需要密码, 则赋值为 NULL
//                  如果需要密码, 则赋值密码串
//-----
{
    //-----
    // 声明并初始化局部变量

    FILE *hSource;
    FILE *hDestination;

    HCRYPTPROV hCryptProv;
    HCRYPTKEY hKey;
    HCRYPTHASH hHash;

    PBYTE pbKeyBlob = NULL;
    DWORD dwKeyBlobLen;

    PBYTE pbBuffer;
    DWORD dwBlockLen;
    DWORD dwBufferLen;
    DWORD dwCount;

    BOOL status = FALSE;

    //-----
    // 打开源文件
    if(!(hSource = fopen(szSource,"rb")))
    {
        HandleError("Error opening ciphertext file!");
    }
    //-----
    // 打开目标文件

    if(!(hDestination = fopen(szDestination,"wb")))
    {
```

```

        HandleError("Error opening plaintext file!");
    }
    //-----
    // 获取缺省提供商句柄
    // 使用笔者个人的密码容器
    // 读者在自己的机器上运行本函数时, 请
    // 使用自己的密码容器名

    if(!CryptAcquireContext(
        &hCryptProv,
        "黄军",
        NULL,
        PROV_RSA_FULL,
        0))
    {
        HandleError("Error during CryptAcquireContext!");
    }
    //-----
    // 检查密码 (password) 是否存在

    if(!szPassword)
    {
        //-----
        // 用保存的会话密钥 (saved session key) 解密文件

        // 从源文件中读出密钥块长度, 并分配内存
        fread(&dwKeyBlobLen, sizeof(DWORD), 1, hSource);
        if(ferror(hSource) || feof(hSource))
        {
            HandleError("Error reading file header!");
        }
        if(!(pbKeyBlob = (BYTE *)malloc(dwKeyBlobLen)))
        {
            HandleError("Memory allocation error.");
        }
        //-----
        // 从源文件中读出密钥块
        fread(pbKeyBlob, 1, dwKeyBlobLen, hSource);
        if(ferror(hSource) || feof(hSource))

```

```
{
    HandleError("Error reading file header!\n");
}
//-----
// 输入密钥块到 CSP

if(!CryptImportKey(
    hCryptProv,
    pbKeyBlob,
    dwKeyBlobLen,
    0,
    0,
    &hKey))
{
    HandleError("Error during CryptImportKey!");
}
}
else
{
    //-----
    // 用从密码 (password) 中派生出来的会话密钥解密文件

    //-----
    // 创建哈希对象

    if(!CryptCreateHash(
        hCryptProv,
        CALG_MD5,
        0,
        0,
        &hHash))
    {
        HandleError("Error during CryptCreateHash!");
    }
    //-----
    // 在密码数据中进行哈希运算

    if(!CryptHashData(
        hHash,
```

```

        (BYTE *)szPassword,
        strlen(szPassword),
        0))
    {
        HandleError("Error during CryptHashData!");
    }
    //-----
    // 从哈希对象派生会话密钥

    if(!CryptDeriveKey(
        hCryptProv,
        ENCRYPT_ALGORITHM,
        hHash,
        0,
        &hKey))
    {
        HandleError("Error during CryptDeriveKey!");
    }
    //-----
    // 注销哈希对象

    CryptDestroyHash(hHash);
    hHash = 0;
}
//-----
// 现在解密密钥准备好了。或者从读自源文件中的块输入,或者使用密码创建
// 如果解码密钥没有的话, 程序不能运行到这点

//-----
// 确定一次解密的字节数
// 其值必须是 ENCRYPT_BLOCK_SIZE 的倍数

dwBlockLen = 1000 - 1000 % ENCRYPT_BLOCK_SIZE;
dwBufferLen = dwBlockLen;

//-----
// 分配内存

if(!(pbBuffer = (BYTE *)malloc(dwBufferLen)))

```



```
{
    HandleError("Out of memory!\n");
}
//-----
// 解密源文件，并写到目标文件
do {
//-----
// 从源文件中读取 dwBlockLen 字节的内容

dwCount = fread(
    pbBuffer,
    1,
    dwBlockLen,
    hSource);
if(ferror(hSource))
{
    HandleError("Error reading ciphertext!");
}
//-----
// 解密数据
if(!CryptDecrypt(
    hKey,
    0,
    feof(hSource),
    0,
    pbBuffer,
    &dwCount))
{
    HandleError("Error during CryptDecrypt!");
}
//-----
// 写数据到目标文件

fwrite(
    pbBuffer,
    1,
    dwCount,
    hDestination);
if(ferror(hDestination))
```

```
{
    HandleError("Error writing plaintext!");
}
}
while(!feof(hSource));
status = TRUE;

//-----
// 关闭文件
if(hSource)
    fclose(hSource);
if(hDestination)
    fclose(hDestination);

//-----
// 释放内存。

if(pbKeyBlob)
    free(pbKeyBlob);
if(pbBuffer)
    free(pbBuffer);

//-----
// 注销会话密钥

if(hKey)
    CryptDestroyKey(hKey);

//-----
// 注销哈希对象
if(hHash)
    *CryptDestroyHash(hHash);

//-----
// 释放提供商句柄

if(hCryptProv)
    CryptReleaseContext(hCryptProv, 0);
```

```

    return status;
} // Decryptfile 结束

```

读者可以修改本程序，设置自己喜欢的加密算法，Microsoft CryptoAPI 中提供了很多加密算法，不过本程序只使用了 3DES 算法对数据进行加密、解密。

下面给出 Windows Server 2000 下的 CryptoAPI 的一些相关参数，以供读者参考。表 10.2 列出了可用的提供商类型 (Provider Type)。

表 10.2 可用的提供商类型 (Provider Type)

提供商类型 (Provider Type)	提供商类型名 (Provider Type Name)
1	RSA Full (Signature and Key Exchange)
3	DSS Signature
12	RSA Schannel
13	DSS Signature with Diffie-Hellman Key Exchange
18	Diffie-Hellman Schannel

表 10.3 列出了可用的提供商 (Provider)。

表 10.3 可用的提供商 (Provider)

提供商类型 (Provider Type)	提供商名 (Provider Name)
1	Gemplus GemSAFE Card CSP v1.0
1	Microsoft Base Cryptographic Provider v1.0
13	Microsoft Base DSS and Diffie-Hellman Cryptographic Provider
3	Microsoft Base DSS Cryptographic Provider
18	Microsoft DH Schannel Cryptographic Provider
1	Microsoft Enhanced Cryptographic Provider v1.0
13	Microsoft Enhanced DSS and Diffie-Hellman Cryptographic Provider
5	Microsoft Exchange Cryptographic Provider v1.0
12	Microsoft RSA Schannel Cryptographic Provider
1	Microsoft Strong Cryptographic Provider
1	Schlumberger Cryptographic Service Provider

缺省的提供商为 Microsoft Strong Cryptographic Provider。表 10.4 列出了支持的算法 (supported algorithms)。

表 10.4 支持的算法 (supported algorithms)

算法的 ID (Algid)	位 (Bits)	类型 (Type)	名字长度 (Name Length)	算法名 (Algorithm Name)
00006602h	40	Encrypt	4	RC2
00006801h	40	Encrypt	4	RC4
00006601h	56	Encrypt	4	DES
00006609h	112	Encrypt	13	3DES TWO KEY
00006603h	168	Encrypt	5	3DES
00008004h	160	Hash	6	SHA-1
00008001h	128	Hash	4	MD2
00008002h	128	Hash	4	MD4
00008003h	128	Hash	4	MD5
00008008h	288	Hash	12	SSL3 SHAMD5
00008005h	64	Hash	4	MAC
00002400h	512	Signature	9	RSA_SIGN
0000a400h	512	Exchange	9	RSA_KEYX
00008009h	0	Hash	5	HMAC

10.8.2 Delphi 例子

下面给出在 Delphi 5.0 下调用上面生成的 EncryptService.dll。

项目创建步骤如下。

- (1) 首先创建一个新的目录用于存储新工程，目录建议命名为 Call DLL。
- (2) 生成一个工程，命名为 project。
- (3) 编辑相应的文件。

下面给出的图 (Delphi 项目的主界面如图 10.14 所示) 是项目的主界面。



图 10.14 Delphi 项目的主界面

下面给出源码:

```
unit Main;
//演示如何调用 EncryptService.DLL 中的函数

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Buttons;

type
  TForm1 = class(TForm)
    OpenDialog1: TOpenDialog;
    SaveDialog1: TSaveDialog;
    BitBtn1: TBitBtn;
    Edit1: TEdit;
    Edit2: TEdit;
    BitBtn2: TBitBtn;
    BitBtn3: TBitBtn;
    Edit3: TEdit;
    Label1: TLabel;
    BitBtn4: TBitBtn;
    procedure BitBtn1Click(Sender: TObject);
    procedure BitBtn2Click(Sender: TObject);
    procedure BitBtn3Click(Sender: TObject);
    procedure BitBtn4Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
//声明 DLL
```

```

//加密服务的 DLL
//声明外部函数
//详细调用方法请参见 Delphi 的帮助文档
{extern "C" extern ENCRYPTDLL_API BOOL MyDecryptFile(
    PCHAR szSource,
    PCHAR szDestination,
    PCHAR szPassword);
}

function MyDecryptFile(
    szSource:PChar;
    szDestination:PChar;
    szPassword:PChar):Boolean;stdcall;far external 'EncryptService.dll'

function MyEncryptFile(
    szSource:PChar;
    szDestination:PChar;
    szPassword:PChar):Boolean;stdcall;far external 'EncryptService.dll'

procedure TForm1.BitBtn1Click(Sender: TObject);
begin
    //指定需要加密的文件
    if OpenFileDialog1.Execute then
        begin
            Edit1.Text:=OpenDialog1.FileName;
        end;
end;

procedure TForm1.BitBtn2Click(Sender: TObject);
begin
    //指定已经加密的文件名
    if SaveDialog1.Execute then
        begin
            Edit2.Text:=SaveDialog1.FileName;
        end;
end;

procedure TForm1.BitBtn3Click(Sender: TObject);
var
    szSource:PChar;
    szDestination:PChar;

```

```
    szPassword:PChar;
begin
    //转换数据类型
    szSource:=PChar(Edit1.text);
    szDestination:=PChar(Edit2.Text);
    if Edit3.Text<>" then
        begin
            szPassword:=PChar(Edit3.Text);
        end
    else
        szPassword:=nil;
    //调用加密函数
    MyEncryptFile(szSource,szDestination,szPassword);
end;
procedure TForm1.BitBtn4Click(Sender: TObject);
var
    szSource:PChar;
    szDestination:PChar;
    szPassword:PChar;
begin
    //转换数据类型
    szSource:=PChar(Edit1.text);
    szDestination:=PChar(Edit2.Text);
    if Edit3.Text<>" then
        begin
            szPassword:=PChar(Edit3.Text);
        end
    else
        szPassword:=nil;
    //调用解密函数
    MyDecryptFile(szSource,szDestination,szPassword);
end;

end.
```

本章小结

本章主要讲述了对称密钥加密和公开密钥加密基本概念,并详细介绍了三重 DES 两个密钥的流程图,介绍了数字签名和数字信封的基本概念。接下来讲述了密码应用编程接口(API)工作模式,说明微软信息密码系统。利用 Microsoft Crypto Graphics 提供的函数,介绍了创建签名消息、加密并封装一个消息、校验签名的消息、加密算法源码分析的具体流程图,其中介绍了一些关键函数的使用方法。最后给出了一个例子介绍如何访问 Microsoft CryptoAPI 提供的加密、解密服务。

第 11 章 强大的项目管理工具

Rational Rose

本章主要内容：

- Rose 简介
- Rose Delphi Link 简介
- UML 简介
- Rose 在项目设计和管理中的具体应用

Rational 公司是目前世界上 CASE 领域的领导者。它紧跟软件科学发展的主流，在软件工程化思想日益深入和面向对象技术（OOA、OOD、OOP）逐步完善和成熟的背景下，提出自动的、科学的、基于组件的软件开发模式并提供相应的理论、技术和产品，覆盖了软件工程化的各个环节（需求管理，可视化建模，软件质量自动保证、过程自动化、软件配置管理和开发工具等，如图 11.1 所示）。

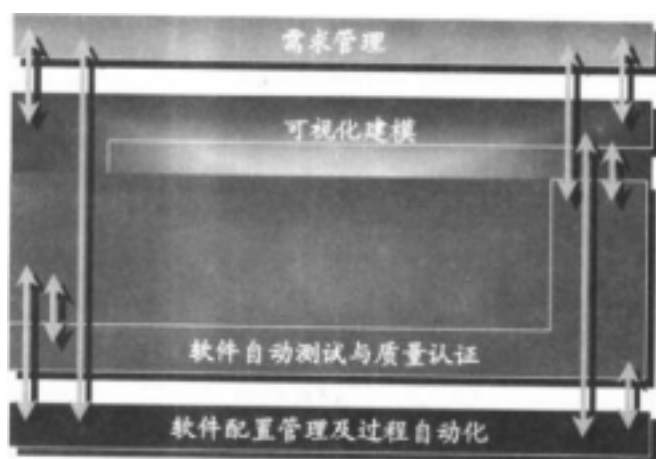


图 11.1 Rational 全面解决方案

Rational Rose 是 Rational 全面解决方案的一个软件模块，一个可视化的面向对象分析和设计的建模工具，为项目提供可视化建模，适合于管理项目开发的需求分析、概要设计、详细设计。

本章首先介绍 Rose 的基本情况，详细介绍 Rose Delphi Link 在 Rose 模型和 Delphi 项目中的应用，然后介绍 Rose 支持的 UML 的基本元素，UML 在项目不同的阶段作用，最后给出 Rational Rose 在项目设计中的具体应用。

11.1 Rose 简介

一般来说,与组件联系最为紧密的是可视化建模工具和自动测试工具。Rational 公司的最新产品 Rational Rose 2001 (以下简称 Rose) 是一个可视化的面向对象分析和设计的建模工具,可对系统需求、业务处理过程、业务对象、软件组建、系统结构和对象进行可视化建模,使应用更贴近商业需求;它支持微软的三层体系结构模型 (Microsoft Three-Tiered Model)、统一建模语言 (UML) 和逆向 (反向)/双向工程 (Reverse/Round-Trip Engineering);它共分 4 层模型,即用例 (Use Case) 模型,逻辑 (Logical) 模型 (类和对象模型),组件 (Component) 模型和部署 (Deployment) 模型;另外,它与多种开发语言紧密继承 (Visual Basic、Visual C++、C++、Java、Ada、SmallTalk、Forte、Delphi 等),支持关系数据库逻辑模型的生成,包括 Oracle、Sybase、SQL Server、Watcom SQL 和 ANSI SQL 等;它可运行在 Windows 95/NT 和各种 Unix 平台上。

Rational Rose 2001 的主界面如图 11.2 所示,其中的标记方法为 UML。

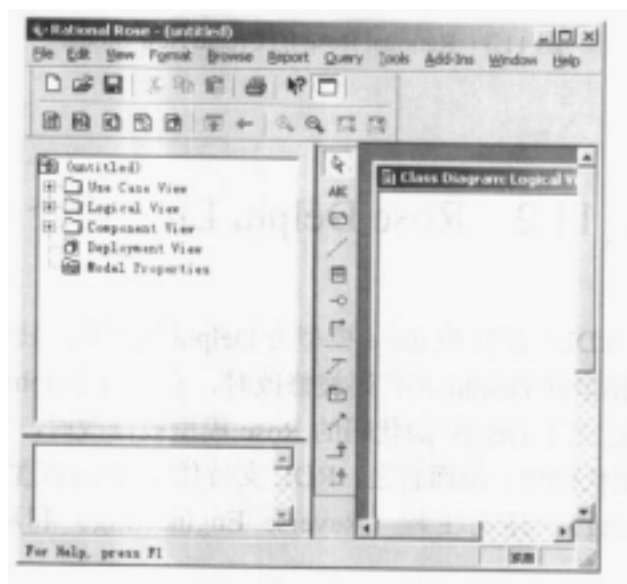


图 11.2 Rational Rose 2001 主界面

Rose 的设计基于面向对象的技术。Rose 提供静态/动态的逻辑视图和物理视图,来帮助用户提取在 OOA 和 OOD 过程中的结果。采用某种标记方法 (Notation),Rose 使得用户在一个表示问题域和软件系统的统一模型 (Overall Model) 中创建和修改各种视图。统一模型由各种模型元素 (类、用例、对象、逻辑包、操作、组件包、组件、处理器和设备等) 以及各元素之间的关系组成。为了可视化地操作这些模型元素及其关系,模型中采用了图 (Diagrams) 和规格说明 (Specifications)。

对应于不同的标记方法 (UML/OMT/Booch),每个模型元素、它的属性以及它们之间的关系都有相应的图标 (Graphic Icons) 表示。Rose 提供相应的工具使得用户对视图、模型元素及其属性和关系拥有全面而灵活的控制。为了节省篇幅,本章主要讲述 UML 标记方法。

Rational Rose 的 Create New Modal 如图 11.3 所示。

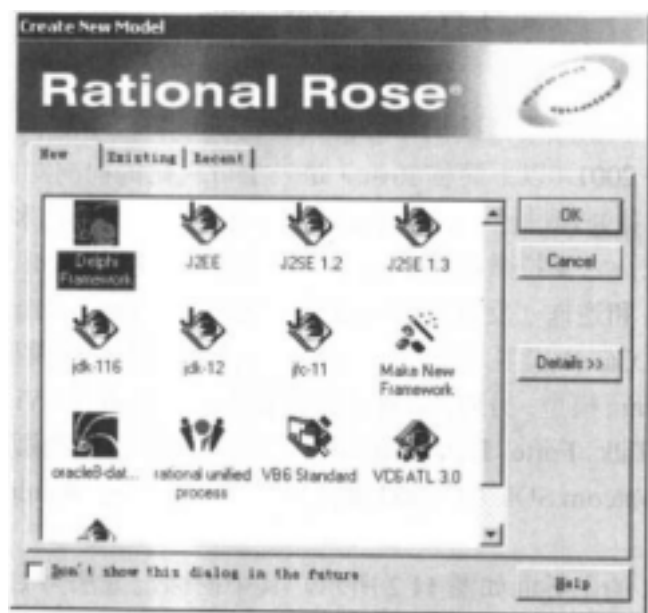


图 11.3 Rational Rose 的 Create New Modal

11.2 Rose Delphi Link 简介

Rose Delphi Link (RDL) 能转换 Rose 模型为 Delphi 源代码, 转换 Delphi 源代码为 Rose 模型。当对应于 Rose 模型的 Delphi 源代码被修改时, 它能将 Delphi 源代码的修改更新到已有的 Rose 模型中; 当对应于 Delphi 源代码的 Rose 模型被修改时, 它能将 Rose 模型的修改更新到已有的 Delphi 源代码中。简而言之, RDL 支持使用 Rose 模型的 Delphi 项目的正向工程 (Forward Engineering)、逆向工程 (Reverse Engineering) 和双向工程 (Round-Trip Engineering)。

1. Delphi 的兼容性

RDL 在 Rational Rose 98、Rose 98I、和 Rose 2000 for Windows 下工作。Rational Rose 的评估拷贝可在 <http://www.rational.com/> 找到。

RDL 设计用来产生 Delphi 5.0 的源代码和从 Delphi 5.0 源代码产生 Rose 模型。它与早期的 Delphi 版本也高度兼容。只要不设定特定于该 Delphi 版本的代码生成特性, 就能产生被早期 Delphi 版本编译的代码。除了在 Delphi 版本之间改变的语言元素 (Language Element), RDL 能够对早期版本中写的项目进行反向工程。

正确的 RDL 发布版本, 可通过下面的操作获得, 即在 RDL 的主浏览器窗口的 Help 菜单选择 About 命令。如图 11.4 所示。

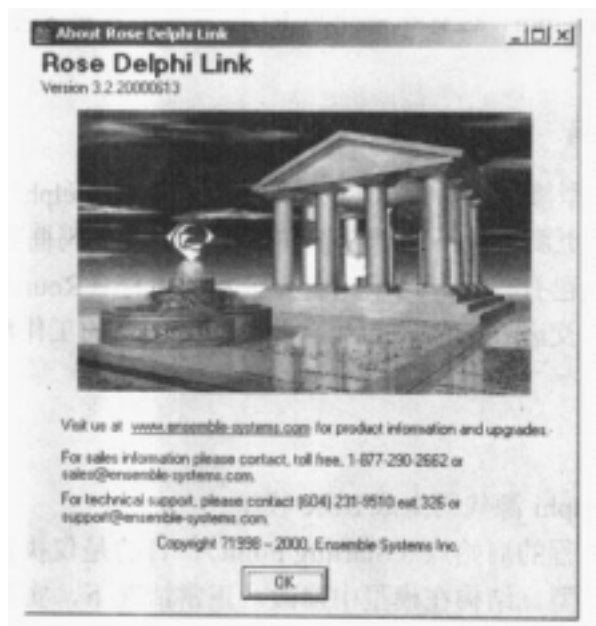


图 11.4 RDL 的 About 窗口图

2. 平台兼容性

RDL 软件可运行的平台如下：运行于 486 或者 Pentium 以上计算机的 Windows 9x、Windows NT 4.0（带 SP3 或更高版本）、或者 Windows 2000。Rose Delphi Link 的最少需求与 Delphi 和 Rose 的相同。

3. 安装

RDL 可从 <http://www.ensemble-systems.com> 下载，或者包含在 Ensemble 的 CD 套件中。RDL 安装文件用 InstallShield 创建。它自动安装软件到机器上，用 RDL 要求的路径更新注册表。

11.2.1 RDL 的操作原则

1. 在模型中没有实现代码

从 Rose 模型生成 Delphi 源代码，或从 Delphi 源代码生成 Rose 模型。在 Rose 模型和 Delphi 源代码之间传输的信息仅包括能被模型指定的信息。这意味着从模型中生成的 Delphi 源代码是一个框架。当用代码更新模型时，主体（Body-Code）并不出现在模型中，而仅改变模型部分的内容。类似地，在模型改变之后，代码被更新，这些改变影响代码结构，而不影响代码体（Body-Code）或者任何其他的未在模型中反映的代码。

2. Delphi 代码生成

使用 RDL 可以用含在 Rose 模型中的信息生成 Delphi 源代码。每一个选择的模型组件（Model Component）生成的代码是一个组件 Rose 规格说明和代码生成特征（Code Generation Properties CGPs）值的函数。这些特征提供了映射模型到 Delphi 源代码的特定于语言的信息。

注意: RDL 不使用 Delphi, 这意味着使用 RDL 不需要在机器上安装 Delphi, 不过, 需要安装 Rose。

3. 更新 Delphi 源代码

RDL 允许在 Rose 模型修改之后用 Rose 模型更新已有的 Delphi 源代码。更新代码的观点继承于双向工程的观点。更新代码不同于在正向工程中生成代码框架 (Code Framework), 它的目标是包含代码主体的已有代码。这意味着为了双向工程 (Round-Trip Engineering) 的目的, 模型和源代码永远是交融在一起, 相互重叠, 却是不同的工作域 (Distinct Domains Of Work)。

4. Rose 模型生成

使用 RDL 能够用 Delphi 源代码生成 Rose 模型。

主要用来作为双向工程的起始点 (Starting Point), 目的是仅执行反向工程一次, 然后为以后的代码生成而使用模型。结构在模型中修改。正常情况下, 实现代码应在 Delphi 的 pas 文件中修改。

注意: 反向工程并不从源代码传输所有的信息到模型中。这意味着, 当从由反向工程创建的模型再生成 (更新) 代码时, 必须将原有项目设定为目标。

反向工程的第二个用途是使用源代码互斥地说明或者分析生成地代码。

在源代码中, 为每一个选择的类生成的模型取决于那个类的 Delphi 类接口规定、它的注释和包含代码的目录和文件。

请注意以下几点:

- Delphi 的特性 (Property) 反向工程到 Rose 中作为操作, 其模板 (Stereotype) 设为 Property。

- RDL 仅根据类间关系 (Interclass Relationship) 定义 “Uses” 关系。

在从代码生成模型之后, 模型可以被修改。

5. 用代码更新 Rose 模型

因在模型中进行结构的更改, 所以模型不太可能根据代码进行更新。不过, 在修改了源代码中的结构 (比如增加一个新类或者删除一个类) 之后, RDL 提供了更新模型的功能。当在源代码中增加新类, 并更新 Rose 模型时, RDL 保留已在模型中的类, 同时将删除了的 (或者更名了的) 类移到一个特定的包 (Special Package) 中。

11.2.2 使用 Rose Delphi Link

根据 Rose Delphi Link 的安装, 子菜单标题 Ensemble Tools 出现在 Rose Tools 菜单中。它包括标题 Rose Delphi Link。此菜单标题启动 RDL 浏览器 (如图 11.5 所示), 它是 RDL 创建和更新操作的出发点。



图 11.5 RDL 浏览器图

RDL 和当前装载的 Rose 模型一起工作。一个模型可能有几个相关的项目（即一个主程序加上 DLL 或者一系列分布的主程序）。为模型创建的 Delphi 项目在 Rose 模型组件视图（Component View）中显示为组件。RDL 浏览器一次只能处理一个项目。

1. RDL 浏览器

为当前 Rose 中装载的模型创建一个新的 RDL 项目：

- （1）在 RDL 浏览器中选择 File 菜单中的 New Project 命令。下面弹出目录浏览器（Directory Browser）。
- （2）浏览到希望创建的 Delphi 项目所在的目录。默认情况下，项目创建在 Rose 模型所在的目录。生成的源代码的目录结构由组件视图（Component View）、而非逻辑视图（Logical View）的包结构确定。
- （3）输入 Delphi 项目名。
- （4）单击 Open 按钮，出现如图 11.6 所示的对话框。

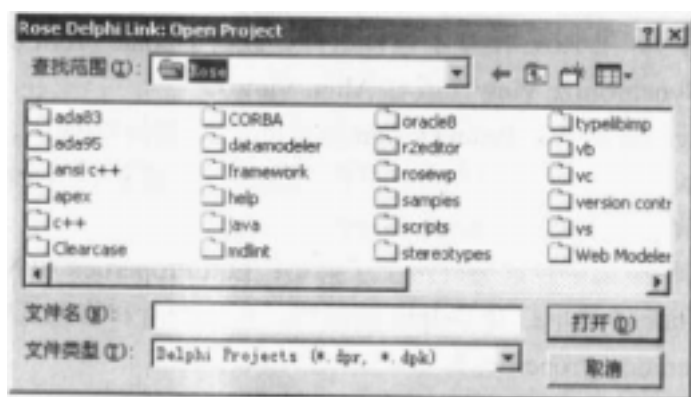


图 11.6 打开项目

新项目作为一个组件（在组件视图中）出现在 Rose 模型中。打开一个已有的 RDL 项目，

即在 RDL 浏览器的 File 菜单中选择 Open Project 命令, 然后选择并打开项目。当项目被创建或者打开时, RDL 显示一个活动日志, 如果没有错误, 则该窗口自动关闭。活动日志消息框如图 11.7 所示 (在下一页)。

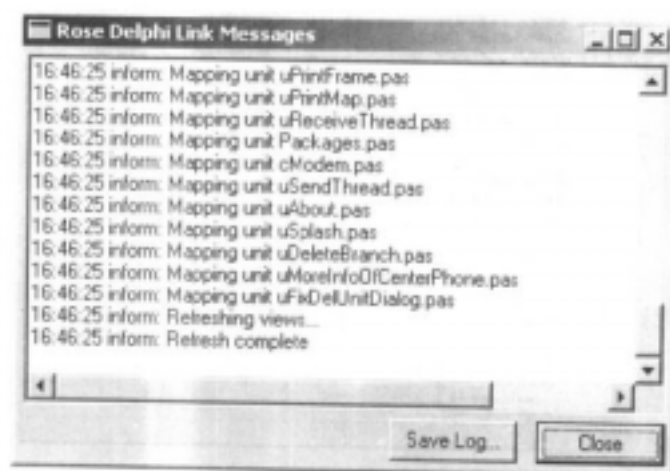


图 11.7 活动日志消息框 (活动日志消息框)

在 View 菜单中单击 Messages 命令可以重新打开这个消息框。

Rose Delphi Link 浏览器显示了两棵树, 一个在左, 一个在右。左边面板给出了当前装载的 Rose 模型的组件视图, 右边面板给出相应的 Delphi 源代码的组件视图。

在 Rose 模型中, 只有分配到相应的组件的类, 即能生成代码的类, 才出现在浏览器中。

实体标记 (Solid Item) 表示该组件存在。虚标记 (Dotted Item) 表示组件不存在, 相应于另一端的实体标记。

Rose 模型和 Delphi 源代码的差异用红色惊叹号表示。浏览器底端的状态条给出在浏览器中选中条目的详细值。

如果源代码或者模型修改了, 可用 Refresh 按钮更新显示结果。单击 Refresh 按钮, RDL 显示活动日志。

Update All 按钮用来创建或者更新从已有代码到整个模型 (对应于 Rose 模型树下的按钮) 或者从已有 Rose 模型到整个 Delphi 项目 (对应于 Delphi 源代码树下的按钮)。

在 Delphi 源代码树中鼠标右键单击一个条目 (Item), 打开一个菜单, 包含如下内容, 即 Create From Design (如果 Rose 模型不包含组件) 或者 Update From Design (如果 Rose 模型已包含组件), 和 Synchronize View (或者 Align View)。

Create From Design 和 Update From Design 用来创建/更新模型/代码。

Synchronize View 用来在同一相应点打开两边的树。要察看另一个树上的相应点, 右键单击该点, 从快捷菜单中选择 Synchronize View。

Rose 模型树中的快捷菜单包含另外两个子菜单: Edit Properties 和 New。

(1) Edit Properties 子菜单打开 Delphi 编辑器, 允许查看和修改选中的模型元素的代码生成特性 (Code Generation Properties)。

(2) New 子菜单允许在 RDL 浏览器中创建新的 Rose 模型元素。这个子菜单的内容取决于所选中的模型元素。新生成的元素创建在选中的元素下, 因此此菜单仅包含选中节点的合法子节点。New 子菜单的菜单条图如图 11.8 所示。

一旦元素创建, Delphi 编辑器将出现, 并允许修改该元素的细节。使用 Rose Delphi Link 可能涉及以下项目。

- (1) 从 Rose 模型生成一个新的 Delphi 项目。
- (2) 从 Delphi 源代码生成一个新的 Rose 项目。
- (3) 从 Rose 模型更新 Delphi 源代码。
- (4) 从 Delphi 源代码更新 Rose 模型。



图 11.8 New 子菜单的菜单条

在所有这些情况下, 浏览器会显示 Rose 和 Delphi 树中所有存在的 (或潜在的) 组件。差别仅取决于哪个组件存在、哪个组件不存在和到底要求 RDL 做什么。

因此使用 RDL 仅意味着整体地或者部分地在一个方向或者另一个方向协调模型和代码。

2. 生成 Delphi 源代码

为一个节点和它的下级节点生成 Delphi 代码。

- (1) 在 Delphi 树中鼠标右键单击节点。
- (2) 在上下文菜单中选择 Create From Design。

整个 Rose 模型生成 Delphi 源代码的两种方法如下。

- ☐ 在树的根部右键单击项目名, 选择 Create From Design。
- ☐ 单击 Delphi 代码树下的 Update All 按钮。

生成的 Delphi 源代码出现在由 Rose 组件视图包结构派生的目录结构中。Delphi 项目文件的位置用作这个目录结构的起始点。生成的代码可以根据需要打开和修改。使用一个简单的文本编辑器 (或者 Delphi IDE) 可以增加实现代码。

3. 生成一个 Rose 模型

为单一节点和下级节点生成 Rose 模型元素。

- (1) 在 Rose 树中右键单击节点。
- (2) 在上下文菜单中选择 Create From Code。

所有的 Delphi 源代码生成一个 Rose 模型的两种方法如下。

- ❑ 在树的根部右键单击模型名，选择 Create From Code。
 - ❑ 单击 Rose 树下的 Update All 按钮。
- 生成 Rose 模型时的活动日志窗体如图 11.9 所示。

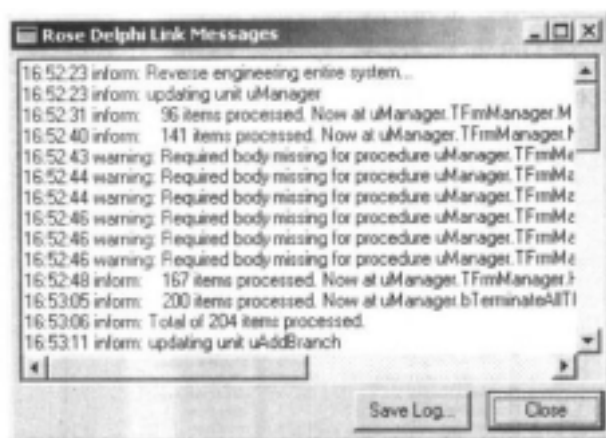


图 11.9 生成一个 Rose 模型

组件的分配由 RDL 处理，即分配类到组件。模型生成之后可以修改分配，然后再重新生成代码。

注意：如果移动一个已有的类，在新的位置仅生成一个框架。实现代码必须人工移动。

生成的 Rose 模型元素增加到当前装载在 Rose 内的模型中。在 Rose 组件视图和逻辑视图中，生成模型的包结构像 Delphi 项目的目录结构。如果源代码引用不在最终目标模型的任何类，Rose 模型将包含“Unresolved References”的包。这主要是 Delphi 库类。

注意：RDL 创建分类“ClassDiagram”特性描述的图表 (Diagram)。如果非“”，在反向工程中，分类中创建的类加到特定的图表。为了确保内容适合于读者的偏好，结果图表很可能需要人工编辑。

当进行反向工程时，记住以下几个要点。

- ❑ RDL 不读“include”文件。
- ❑ RDL 不解释{\$IFDEF}编译器指示 (Compiler Directives)，它们均被假定为“False”。
- ❑ RDL 要求分号，而 Delphi 4 编译器认为是可选的。
- ❑ 在 Rose 模型中属性的初始值不设定，因为在 Delphi 中类属性不能有初始值。
- ❑ Delphi 的方法主体不存在 Rose 模型中。

❑ RDL 发现并读入在 dpr 文件中列出的所有单元，和被这些单元“使用”的所有单元，当且仅当那些单元能在项目搜索路径中找到。RDL 不使用库搜索路径查找单元。如果真的需要反向工程不在项目搜索路径的单元，将该单元增加到项目，或者编辑 dpr 文件，或者临时增加位置到项目搜索路径中。设置项目搜索路径有两种方法：

- (1) 在 Delphi 菜单 Project 中单击 Options...，然后选择 Directories/Conditionals;
- (2) 编辑项目名.dof 文件。

❑ 当进行反向工程一个模型元素时，其父元素不存在，RDL 自动反向工程足够的父元素根版本 (Stub Version) 以创建一个一致的模型。不过，父元素可能需要完全反向工程。

4. 更新 Rose 模型和 Delphi 源代码

当 Rose 模型和相应的 Delphi 源代码均存在时, RDL 可以保持两者同步。读者甚至可以同时修改代码和模型, 只要修改发生在不同部分。

为了协调 Rose 模型和 Delphi 源代码之间的差异, 要进行如下操作。

(1) 在想修改的树中选择希望协调的元素, 即目标。如果希望 Rose 模型反映 Delphi 源代码的最新情况, 在 Rose 模型树中选择相关的元素, 如图 11.10 所示。



图 11.10 协调 Rose 模型和 Delphi 源代码之间的差异

(2) 右键单击该元素, 在弹出的快捷菜单中选中 Create/Update from design/code 命令。

在正向删除 (Forward Deletion) 中 (即一个元素出现在 Delphi 源代码, 而不在 Rose 模型的情况下, 从 Rose 模型更新 Delphi 源代码), Delphi 源代码中的元素被编译器指示 {\$IFDEF DELETED} 注释掉, 不再出现在 RDL 浏览器中。

在反向删除 (Reverse Deletion) 中 (即一个元素出现在 Rose 模型, 而不在 Delphi 源代码的情况下, 从 Delphi 源代码更新 Rose 模型), Rose 模型中的元素移到 “Deleted Items” 分类, 不再出现在 RDL 浏览器中。

注意: 如果代码和模型均存在, 当使用 Update All 按钮时必须注意。根据更新的方向, 所有标记的差异都包含在代码或者模型中。一定要慎重! 如果需要在两边都作修改, 并且两边的改变都想保留的话, 切记不要使用 Update All 按钮。

11.2.3 修改 RDL 的代码生成特性

在 Rose 中, 类和类成员的代码生成特性 (Code Generation Properties) 可以使用规格说明编辑器 (Specification Editor) 进行修改。它们也能使用 Rose Delphi Link 的特性编辑器 (Property Editor) 进行修改。RDL 编辑器可以从 Rose 或者从 RDL 浏览器中激活。

在 Rose 中激活 RDL 特性编辑器, 可进行如下操作:

- (1) 在类图或者 Rose 浏览器中右键单击需要的模型元素;
- (2) 从弹出的快捷菜单中选择 Delphi Properties..., 或者在 Rose 的 Tools 菜单中选择

Ensemble Tools, 然后选择 Delphi Properties Editor 命令。

从 RDL 浏览器中激活 RDL 特性编辑器, 可进行如下操作:

- (1) 在 Rose 模型中, 右键单击一项;
- (2) 从弹出的快捷菜单中选择 Edit Properties。

RDL 特性编辑器是上下文敏感的, 并与选中的模型元素一致。在编辑器中选项卡的数和内容根据选中的元素而异。然而, Code Preview 选项卡 (Tab) 总是会有。这个选项卡对选中模型元素的将要生成的 Delphi 源代码提供只读预览。Code Preview 选项卡如图 11.11 所示。

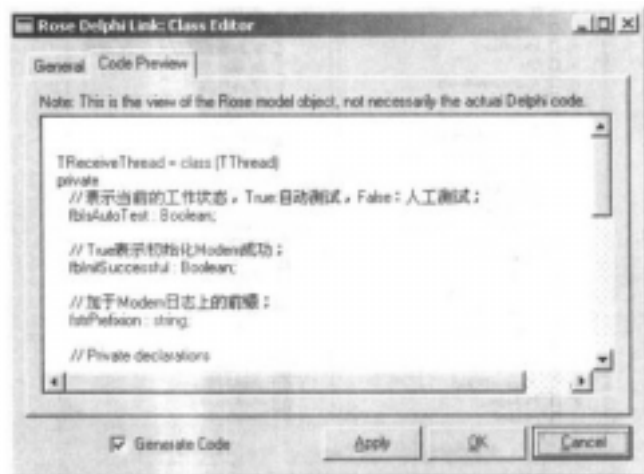


图 11.11 Code Preview 选项卡

注意: 如果应用了修改, 在编辑器中的任何修改仅出现在 Code Preview 选项卡中。

Apply 接受修改而不必关闭编辑器。单击 OK 按钮接受修改, 同时关闭编辑器。

1. 编辑类

Class 编辑器允许用户查看或修改在 General 选项卡中选中类的 Name、Kind、Visibility、Documentation。类编辑器的 General 选项卡如图 11.12 所示。

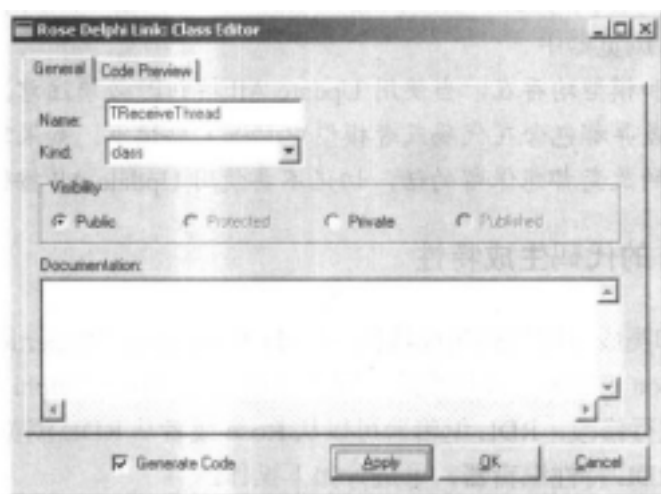


图 11.12 类编辑器的 General 选项卡



图 11.13 组件编辑器的 General 选项卡

这个类的类型可设为 Class、Interface、Dispinterface，或者 Object。

2. 编辑组件

控件编辑器允许用户查看或修改在 General 选项卡中选中控件的 Name、Documentation。这个组件的可视性是无效的。组件编辑器的 General 选项卡如图 11.13 所示。组件编辑器的 Detail 选项卡如图 11.14 所示。

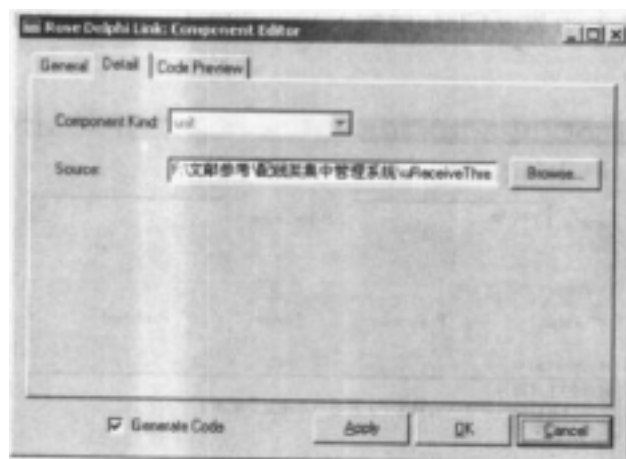


图 11.14 组件编辑器的 Detail 选项卡

在 Detail 选项卡中用户能改变组件的类型和相关的源文件。

这个控件的类型可设为 Unit、Program、Library，或者 Package。

为改变相关的源文件，编辑实体字段（Entry Field）或者用 Browse 按钮选择要求的文件。

3. 编辑属性

属性编辑器允许用户查看或修改在 General 选项卡标签中选中属性的 Name、Type、Visibility、Documentation。属性编辑器的 General 选项卡如图 11.15 所示。



图 11.15 属性编辑器的 General 选项卡

为修改属性的类型，输入新类型或选择在下拉列表（Drop-down List）中的 Delphi 预定义类型。

4. 编辑操作

操作编辑器允许用户查看或修改在 General 选项卡标签中选中 Operation(操作)的 Name、Return 类型(如果操作是个函数或 Class 函数)、Visibility、Documentation。操作编辑器的 General 选项卡如图 11.16 所示。



图 11.16 操作编辑器的 General 选项卡

修改属性的 Operation 类型，输入新类型或选择在下拉列表（Drop-down List）中提供的类型之一。

在标签中用户能修改 Operation、Modifiers、Directives。

Operation 类型或者设为 Destructor、Constructor、Procedure、Function、Class Procedure，或者设为 Class 函数（Class Function）。操作编辑器的 Detail 选项卡如图 11.17 所示。

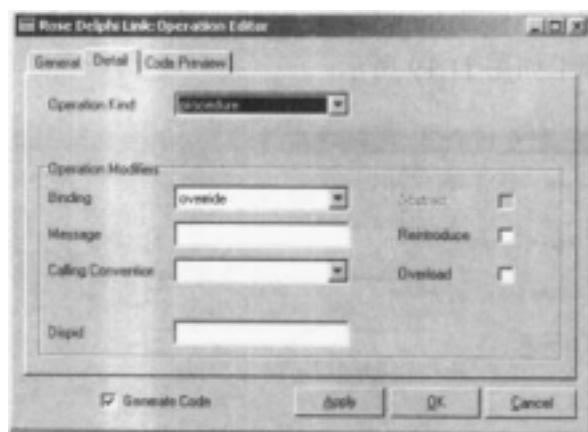


图 11.17 操作编辑器的 Detail 选项卡

5. 编辑特性

特性编辑器(Property Editor)允许用户查看或修改在 General 选项卡中选中特性的 Name、Type、Visibility、Documentation。特性编辑器的 General 选项卡的图略。

为了修改特性类型,输入新类型或选择在下拉列表(Drop-down List)中提供的类型之一。

在 Detail 选项卡中用户能编辑 Properties、Modifiers 和 Directives。特性编辑器的 Detail 选项卡的图略。

如果选择的特性是一个数组特性(Array Property),即它有参数,通过选取数组为默认值,用户能设置是否要设为默认特性。

为修改 Read 或 Write 字段,输入新值,或者从下拉列表(Drop-down List)中提供的方法和特性列表选择一个值。

同样地,修改 Stored 或 Implements 字段,输入新值,或者从下拉列表框表(Drop-down List)中提供的可选项列表选择一个值。

为了输入特性的默认值,首先单击 Default 按钮。

6. 编辑角色

角色编辑器(Role Editor)允许用户查看或修改在 General 选项卡中选中的 Role 的 Name、Visibility、Documentation。角色编辑器的 General 选项卡如图 11.18 所示。

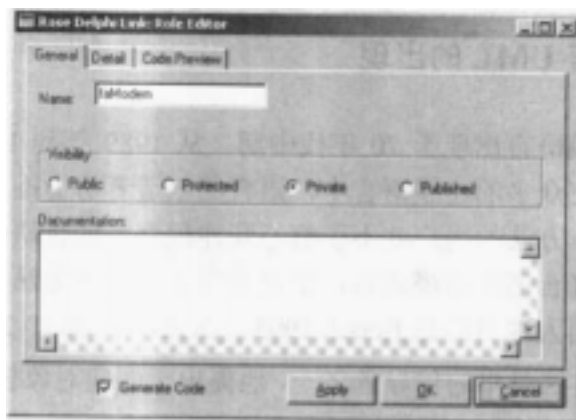


图 11.18 角色编辑器的 General 选项卡

在 Detail 选项卡中用户能修改 Supplier 类型、Supplier 类和 Role 基数 (Role Cardinality)。角色编辑器的 Detail 选项卡如图 11.19 所示。

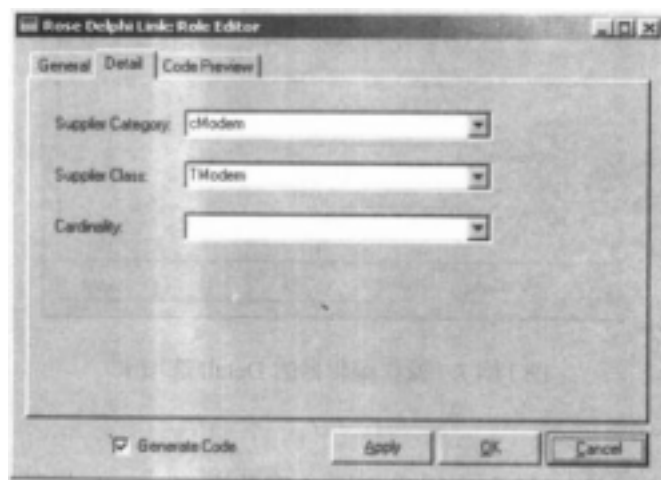


图 11.19 角色编辑器的 Detail 选项卡

为了修改 Supplier 类型、Supplier 类和 Role 基数 (Role Cardinality)，输入新值，或者从下拉列框 (Drop-down List) 中提供的可选项列表选择一个值。

可用的 Supplier 类取决于选择的 Supplier 分类 (Category)。

RDL 特性编辑器提供了丰富的功能对类和类成员的代码生成特性进行修改。具体的操作请读者参考 Rose Delphi Link 提供的帮助。

11.3 UML 简介

面向对象的分析与设计 (OOA&D) 方法的发展到 80 年代末至 90 年代中期时出现了一个高潮，UML 是这个高潮的产物。它不仅统一了 Booch、Rumbaugh 和 Jacobson 的表示方法，而且对其作了进一步的发展，并最终统一为大众所接受的标准建模语言。

11.3.1 标准建模语言 UML 的出现

公认的面向对象建模语言出现于 70 年代中期。从 1989 年到 1994 年，其数量从不到十种增加到了五十多种。在众多的建模语言中，语言的创造者努力推崇自己的产品，并在实践中不断完善。但是，OO 方法的用户并不了解不同建模语言的优缺点及相互之间的差异，因而很难根据应用特点选择合适的建模语言，于是爆发了一场“方法大战”。90 年代中，一批新方法出现了，其中最引人注目的是 Booch 1993、OOSE 和 OMT-2 等。

Booch 是面向对象方法最早的倡导者之一，他提出了面向对象软件工程的概念。1991 年，他把过去面向 Ada 的工作扩展到整个面向对象设计领域。Booch 1993 比较适合于系统的设计和构造。Rumbaugh 等人提出了面向对象的建模技术 (OMT) 方法，采用了面向对象的概念，

并引入各种独立于语言的表示符。这种方法用对象模型、动态模型、功能模型和用例模型，共同完成对整个系统的建模，所定义的概念和符号可用于软件开发的分析、设计和实现的全过程，软件开发人员不必在开发过程的不同阶段进行概念和符号的转换。OMT-2 特别适用于分析和描述以数据为中心的信息系统。Jacobson 于 1994 年提出了 OOSE 方法，其最大特点是面向用例（Use-Case），并在用例的描述中引入了外部角色的概念。用例的概念是精确描述需求的重要武器，但用例贯穿于整个开发过程，包括对系统的测试和验证。OOSE 比较适合支持商业工程和需求分析。此外，还有 Coad/Yourdon 方法，即著名的 OOA/OOD，它是最早的面向对象的分析和设计方法之一。该方法简单、易学，适合于面向对象技术的初学者使用，但由于该方法在处理能力方面的局限，目前已很少使用。

概括起来，首先，面对众多的建模语言，用户由于没有能力区别不同语言之间的差别，因此很难找到一种比较适合其应用特点的语言；其次，众多的建模语言实际上各有千秋；第三，虽然不同的建模语言大多类同，但仍存在某些细微的差别，极大地妨碍了用户之间的交流。因此在客观上，极有必要在精心比较不同的建模语言优缺点及总结面向对象技术应用实践的基础上，组织联合设计小组，根据应用需求，取其精华，去其糟粕，求同存异，统一建模语言。

1994 年 10 月，Grady Booch 和 Jim Rumbaugh 开始致力于这一工作。他们首先将 Booch93 和 OMT-2 统一起来，并于 1995 年 10 月发布了第一个公开版本，称之为统一方法 UM 0.8（Unified Method）。1995 年秋，OOSE 的创始人 Ivar Jacobson 加盟到这一工作。经过 Booch、Rumbaugh 和 Jacobson 三人的共同努力，于 1996 年 6 月和 10 月分别发布了两个新的版本，即 UML 0.9 和 UML 0.91，并将 UM 重新命名为 UML（Unified Modeling Language）。1996 年，一些机构将 UML 作为其商业策略已日趋明显。UML 的开发得到了来自公众的正面反应，并倡议成立了 UML 成员协会，以完善、加强和促进 UML 的定义工作。当时的成员有 DEC、HP、I-Logix、Itellicorp、IBM、ICON Computing、MCI Systemhouse、Microsoft、Oracle、Rational Software、TI 以及 Unisys。这一机构对 UML 1.0（1997 年 1 月）及 UML 1.1（1997 年 11 月 17 日）的定义和发布起了重要的促进作用。

UML 是一种定义良好、易于表达、功能强大且普遍适用的建模语言。它融入了软件工程领域的新思想、新方法和新技术。它的作用域不限于支持面向对象的分析与设计，还支持从需求分析开始的软件开发的全过程。

面向对象技术和 UML 的发展过程可用图 11.20 来表示，标准建模语言的出现是其重要成果。在美国，截止 1996 年 10 月，UML 获得了工业界、科技界和应用界的广泛支持，已有 700 多个公司表示支持采用 UML 作为建模语言。1996 年底，UML 已稳占面向对象技术市场的 85%，成为可视化建模语言事实上的工业标准。1997 年 11 月 17 日，OMG 采纳 UML 1.1 作为基于面向对象技术的标准建模语言。UML 代表了面向对象方法的软件开发技术的发展方向，具有巨大的市场前景。

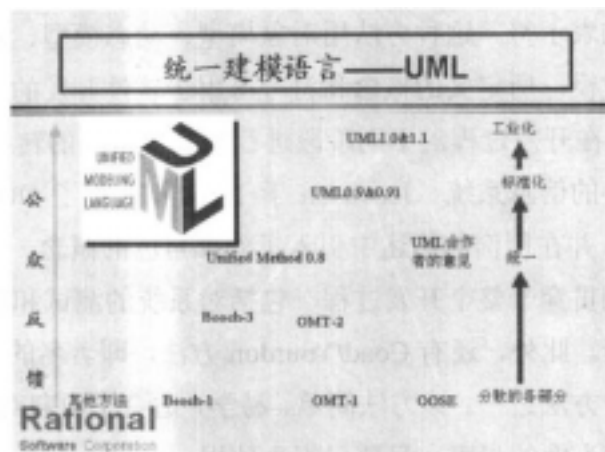


图 11.20 UML 的发展历程

11.3.2 标准建模语言 UML 的内容

首先，UML 融合了 Booch、OMT 和 OOSE 方法中的基本概念，而且这些基本概念与其他面向对象技术中的基本概念大多相同，因而，UML 必然成为这些方法以及其他方法的使用者乐于采用的一种简单一致的建模语言；其次，UML 不仅仅是上述方法的简单汇合，而且还是在这些方法的基础上广泛征求意见，集众家之长，几经修改而完成的，UML 扩展了现有方法的应用范围；第三，UML 是标准的建模语言，而不是标准的开发过程。尽管 UML 的应用必然以系统的开发过程为背景，但由于不同的组织和不同的应用领域，需要采取不同的开发过程。

作为一种建模语言，UML 的定义包括 UML 语义和 UML 表示法两个部分。

1. UML 语义

描述基于 UML 的精确元模型定义。元模型为 UML 的所有元素在语法和语义上提供了简单、一致、通用的定义性说明，使开发者能在语义上取得一致，消除了因人而异的最佳表达方法所造成的影响。此外 UML 还支持对元模型的扩展定义。

2. UML 表示法

定义 UML 符号表示法，为开发者或开发工具使用这些图形符号和文本语法提供了系统建模的标准。这些图形符号和文字所表达的是应用级的模型，在语义上它是 UML 元模型的实例。

标准建模语言 UML 的重要内容可以由下列 5 类图（共 9 种图形）来定义：

□ 第 1 类是用例图，从用户角度描述系统功能，并指出各功能的操作者。

□ 第 2 类是静态图（Static Diagram），包括类图、对象图和包图。其中类图描述系统中类的静态结构。不仅定义系统中的类，表示类之间的联系如关联、依赖、聚合等，也包括类的内部结构（类的属性和操作）。类图描述的是一种静态关系，在系统的整个生命周期都是有效的。对象图是类图的实例，几乎使用与类图完全相同的标识。他们的不同点在于对象图显示类的多个对象实例，而不是实际的类。一个对象图是类图的一个实例。由于对象存在生命周期，因此对象图只能在系统某一段时间段存在。包由包或类组成，表示包与包之间的关系。

包图用于描述系统的分层结构。

□ 第 3 类是行为图 (Behavior diagram), 描述系统的动态模型和组成对象间的交互关系。其中状态图描述类的对象所有可能的状态以及事件发生时状态的转移条件。通常, 状态图是对类图的补充。在实用上并不需要为所有的类画状态图, 仅为那些有多个状态其行为受外界环境的影响并且发生改变的类画状态图。而活动图描述满足用例要求所要进行的活动以及活动间的约束关系, 有利于识别并行活动。

□ 第 4 类是交互图 (Interactive Diagram), 描述对象间的交互关系。其中顺序图显示对象之间的动态合作关系, 它强调对象之间消息发送的顺序, 同时显示对象之间的交互; 合作图描述对象间的协作关系, 合作图跟顺序图相似, 显示对象间的动态合作关系。除显示信息交换外, 合作图还显示对象以及它们之间的关系。如果强调时间和顺序, 则使用顺序图; 如果强调上下级关系, 则选择合作图。这两种图合称为交互图。

□ 第 5 类是实现图 (Implementation Diagram)。其中构件图描述代码部件的物理结构及各部件之间的依赖关系。一个部件可能是一个资源代码部件、一个二进制部件或一个可执行部件。它包含逻辑类或实现类的有关信息。部件图有助于分析和理解部件之间的相互影响程度。配置图 (或称为部署图) 定义系统中软硬件的物理体系结构。它可以显示实际的计算机和设备 (用节点表示) 以及它们之间的连接关系, 也可显示连接的类型及部件之间的依赖性。在节点内部, 放置可执行部件和对象以显示节点跟可执行软件单元的对应关系。

从应用的角度看, 当采用面向对象技术设计系统时, 首先是描述需求; 其次根据需求建立系统的静态模型, 以构造系统的结构; 第三步是描述系统的行为。其中在第一步与第二步中所建立的模型都是静态的, 包括用例图、类图 (包含包)、对象图、组件图和配置图等五个图形, 它们是标准建模语言 UML 的静态建模机制。其中第三步所建立的模型或者可以执行, 或者表示执行时的时序状态或交互关系。它包括状态图、活动图、顺序图和合作图等 4 个图形, 是标准建模语言 UML 的动态建模机制。因此, 标准建模语言 UML 的主要内容也可以归纳为静态建模机制和动态建模机制两大类。

11.3.3 标准建模语言 UML 的主要特点

标准建模语言 UML 的主要特点可以归结为 3 点:

□ UML 统一了 Booch、OMT 和 OOSE 等方法中的基本概念。

□ UML 还吸取了面向对象技术领域其他流派的长处, 其中也包括非 OO 方法的影响。UML 符号表示考虑了各种方法的图形表示, 删掉了大量易引起混乱的、多余的和极少使用的符号, 也添加了一些新符号。因此, 在 UML 中汇入了面向对象领域中很多人的思想。这些思想并不是 UML 的开发者们发明的, 而是开发者们依据最优秀的 OO 方法和丰富的计算机科学实践经验综合提炼而成的。

□ UML 在演变过程中还提出了一些新的概念。在 UML 标准中新加了模板 (Stereotypes)、职责 (Responsibilities)、扩展机制 (Extensibility Mechanisms)、线程 (Threads)、过程 (Processes)、分布式 (Distribution)、并发 (Concurrency)、模式 (Patterns)、合作 (Collaborations)、活动图 (Activity Diagram) 等新概念, 并清晰地地区分类型 (Type)、类 (Class) 和实例 (Instance)、细化 (Refinement)、接口 (Interfaces) 和组件 (Components) 等概念。

因此可以认为,UML 是一种先进实用的标准建模语言,但其中某些概念尚待实践来验证,UML 也必然存在一个进化过程。

11.4 Rose 在项目设计和管理中的具体应用

Rose 支持 3 种标记方法(UML、OMT、Booch)进行软件建模,对应于不同的标记方法(UML/OMT/Booch),每个模型元素、它的属性以及它们之间的关系都有相应的图标(Graphic Icons)表示。Rose 提供相应的工具使得用户对视图、模型元素及其属性和关系拥有全面而灵活的控制。本节采用 UML 讲述 Rose 的软件建模过程。

UML 是一种建模语言而不是方法,这是因为 UML 中没有过程的概念,而过程正是方法的一个重要组成部分。UML 本身独立于过程,这意味着用户在使用 UML 进行建模时,可以选用任何适合的过程。过程的选用与软件开发过程的不同因素有关,诸如所开发软件的种类(如实时系统、信息系统和桌面产品)、开发组织的规模(如单人开发、小组开发和团队开发)等。用户将根据不同的需要选用不同的过程。然而,使用 UML 建模仍然有着大致统一的过程框架,该框架包含了 UML 建模过程中的共同要素,同时又为用户选用与其所开发的工程相适合的建模技术提供了很大的自由度。

11.4.1 UML 建模过程高层视图

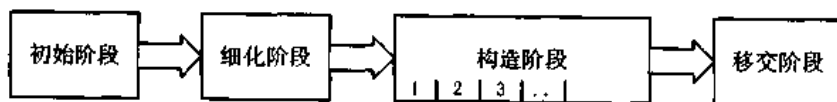


图 11.21 开发过程简图

UML 建模过程的一个高层视图如图 10.21 所示。这是一个迭代递增的开发过程。使用此方法,不是在项目结束时一次性提交软件,而是分块逐次开发和提交。构造阶段由多次迭代组成,每一次迭代都包含编码、测试和集成,所得产品应满足项目需求的某一子集,或提交给用户,或纯粹是内部提交。每次迭代都包含了软件生命周期的所有阶段。同时,每次迭代都要增加一些新的功能,以解决一些新的问题。

因此,首先要做的工作是选择一些功能点,然后完成这些功能,之后再选择别的功能点,如此循环往复。前两个阶段是初始(Inception)和细化(Elaboration)阶段。在初始阶段,需要考虑项目的效益,并确定项目的范围。这一阶段需要与项目出资方进行讨论。在细化阶段,需要收集更为详细的需求,进行高层分析和设计,并为构造阶段制定计划。运用这种迭代开发过程时,还有一些工作(如 β 测试、性能调试和用户培训等)要放到最后的移交阶段(Transition)中进行。

事实上,涉及实际建模工作的微过程存在于上述的每次迭代中。迭代式开发是项目成功的重要保证。

11.4.2 UML 实际建模过程

每次迭代都分为以下几个阶段。

□ 分析阶段：建模的目的是捕捉系统的功能需求，分析、提取所开发系统的“客观世界”领域的类以及描述它们的合作概貌。

□ 设计阶段：建模的目的是通过考虑实现环境，将分析阶段的模型扩展和转化为可行的技术实现方案。

□ 实现阶段：具体工作就是进行编码，同时对已构造的模型作相应的修正。

□ 配置阶段：通过模型描述所开发系统的软硬件配置情况。

□ 测试阶段：使用前几个阶段所构造的模型来指导和协助测试工作。

在系统开发的不同阶段，使用 UML 为系统建模，可以通过建立不同的模型，从不同的视角，以不同的详略程度对系统进行描述。下面以本书前面章节提到的监控系统的开发过程为例，具体介绍 UML 建模的实际过程。

1. 需求

最初版本的集中监控系统的正文需求规格说明应当由代表系统最终用户的人员提供，内容包括系统基本功能需求和对计算机系统和外围设备的要求。大致描述如下。

程控机房的线路、配线架集中告警系统，是由远端检测传输单元和中央监控中心组成。远端检测传输单元安装在配线架机房内，对 MDF 保安单元的告警状态、外线电缆断线等情况进行检测，当发现配线架告警或外线电缆断线时，通过传输单元（Modem）传输到中央监控中心，监控中心声光告警，并通过电话通知相关人员。同时，中央监控中心可以通过 Modem 对远端检测传输单元进行设置。

系统由远端检测传输单元（或称为告警监测仪）和中央监控中心组成。

告警监测仪（监测单元，调解器，采集器）特征参考第 8 章的 8.1.1 节。

监控中心特征参考第 8 章的 8.1.2 节。

由于基于 UML 的系统开发采取增量和迭代方式，集中监控系统的初始版本仅需要完成系统的最基本功能，而其他功能的实现（如语音信息系统等）则在以后的版本中完成。

2. 分析

分析的任务是找出系统的所有需求并加以描述，同时建立模型，以定义系统中的关键领域类，应由系统用户和开发人员合作完成。这一阶段不要拘泥于设计细节和技术方案。

（1）需求分析

分析的第一步是定义用例（Use Case），以描述所开发系统的外部功能需求。用例分析包括阅读和分析需求说明，此时需要与系统的潜在用户进行讨论。用例模型的主要构件是用例、角色和系统边界。用例用于描述每个功能需求，系统边界用于界定系统功能范围，而角色用于描述与系统功能有关的外部实体，它可以是用户，也可以是外部系统。

在本实例中，通过分析，先确认集中监控系统的角色有产品维护人员（客户端的监控进行扩容，增加客户端监控的设备）、查询人员（对数据库进行查询或者打印相关的数据）、超级管理员（可在服务器端发送命令到客户端，可修改数据库内容）。该监控系统有数据查询、数据修改、发送数据、系统维护（包括增加客户端的监控设备数）。如图 11.22 所示。

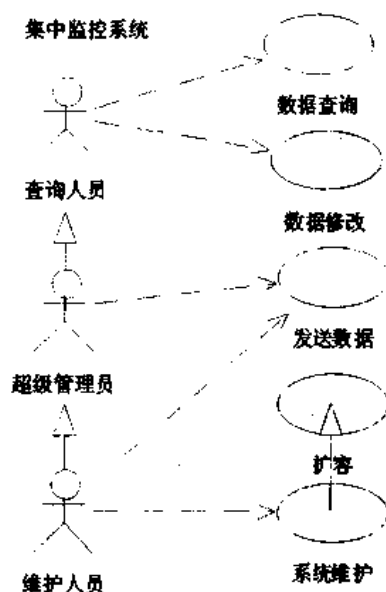


图 11.22 集中监控系统用例图

注意：本节称告警监测仪为客户端，监控中心为服务器端。

除了用例图描述系统需求外，还可以用文字或活动图（Activity Model）对每个用例进行需求说明，更具体地描述该用例与角色的交互。例如可以描述发送数据用例的需求说明如下。

- 生成控制信息包
- 建立连接
- 发送数据

发送数据需求可以用活动图来描述，如图 11.23 所示。由于用例的需求说明直接影响到后续设计阶段对类的操作的定位，因此，用例的需求说明应当尽量全面、准确。发送数据是一个复杂的过程，下面给出生成控制信息图的细节，如图 11.24 所示。

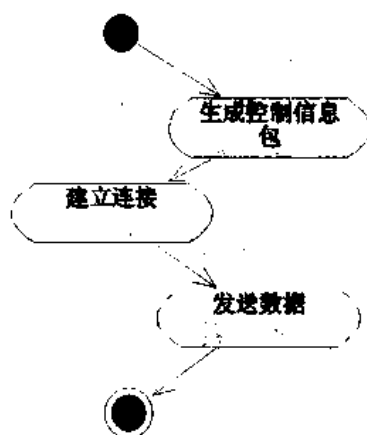


图 11.23 发送数据用例图

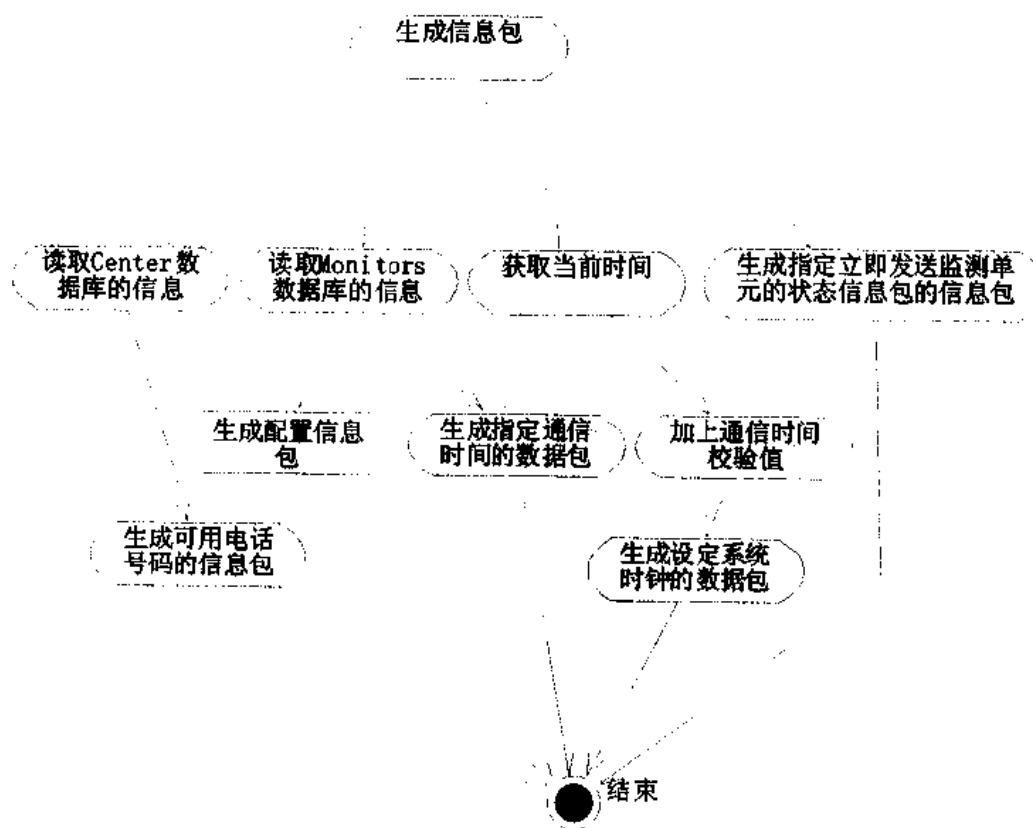


图 11.24 生成控制信息包用例图

值得说明的是，绝大多数用例可以在系统需求分析阶段确定，但随着系统的进展，可能会发现更多的用例，甚至会发现前面定义的用例存在不够确切或错误的地方，需要重新修改。因此，在整个系统开发过程中，都应当时刻关注用例。

(2) 特定领域分析

分析阶段的另一项工作是特定领域分析，以列出系统中的特定领域类。我们可以通过阅读规格说明、用例以及寻找系统处理的“概念”来进行特定领域分析，也可以通过用户和领域专家的讨论，以识别出要处理的所有关键类及它们的相互关系。这里的特定领域是指具体的商业领域，而不是整个系统领域。

在本实例中，可以确定集中监控系统中的特定领域类为设备、监控、数据查询、报警，并使用类图来描述系统领域类及其关系。

需要强调的是，这一阶段对特定领域类的描述具有一定的特殊性，也就是说特定领域类的操作和属性不一定与最终实现时的定义一致。因为此时还没有涉及到系统功能的具体实现，不可能准确、完整地定义它们。有一些操作需要在设计阶段细化时才能确定。

此外，为了描述领域类的动态行为，可以使用 UML 中的任何一种动态图（如顺序图、活动图、合作图、状态图）。本阶段的各动态图都具有素描性质，主要是为了协助对领域类及

其相互关系的分析，为下一阶段的具体设计打下基础。

UML 建模是很灵活的过程，使用者不必面面俱到地画出各种图。对于每一幅图，只有在必要时（比如能帮助分析、设计、指导编码、加深理解、促进交流等）才需要画出，这样的图对建模才有意义，否则会浪费精力而事倍功半。

3. 设计

设计阶段的任务是通过综合考虑所有的技术限制，以扩展和细化分析阶段的模型。设计的目的是指明一种易转化成代码的工作方案，是对分析工作的细化，即进一步细化分析阶段所提取的类（包括其操作和属性），并且增加新类以处理诸如数据库、用户接口、通信、设备等技术领域的问题。

设计阶段可以分为两个部分：结构设计是高层设计，其任务是定义包（子系统），包括包间的依赖性和主要通信机制。我们希望得到尽可能简单和清晰的结构，各部分之间的依赖尽可能的少，并尽可能的减少双向的依赖关系。

第二部分是详细设计，细化包的内容，使编程人员得到所有类的一个足够清晰的描述。同时使用 UML 中的动态模型，描述特定情况下这些类的实例之间的行为。

（1）结构设计

一个设计良好的系统结构是系统可扩充和可变更的基础。类图中包括有助于用户从技术逻辑中分离出应用逻辑（领域类），从而减少它们之间的依赖性。这就是软件结构设计强调的模块间的高聚合、低耦合的原则。

（2）详细设计

详细设计的目的是通过创建新的类图、状态图和动态图，描述新的技术类，并扩展和细化分析阶段“素描”的商业对象类。这些图在分析阶段也曾用过，不过在详细设计阶段，它们是从技术层次上对系统进行更详尽的描述。如分析阶段的用例描述用来验证它们是否在设计阶段都得到处理，而顺序图用来展示系统中每个用例在技术上如何实现等。

在设计阶段，也可细化分析阶段的状态图，更详细的显示状态的变换细节。使用状态图可以揭示单个对象在整个系统中的变化细节，对了解和实现关键类有较大的帮助。

此外，还可以使用其他图在实现层上从不同侧面对分析阶段建立的模型进行细化。

4. 实现

构造或实现阶段是对类进行编程的过程。可以选择某种面向对象编程语言（如 Java、Delphi）作为实现系统的软件环境。Delphi 很容易实现从逻辑视图到代码部件的映射，因为类到 Delphi 代码文件之间是一一映射关系。设计模型的部件图如图 11.25 所示，显示逻辑视图到部件视图的一个简单映射。逻辑视图中的包也映射到相应的部件视图中。

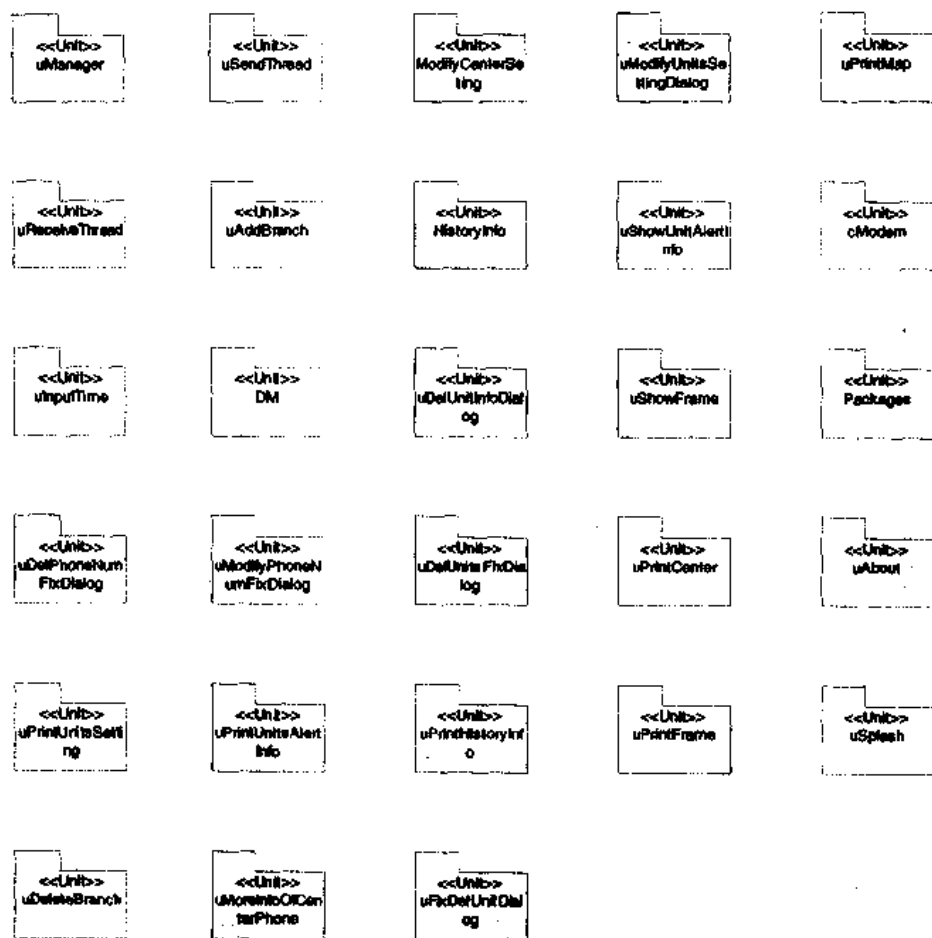


图 11.25 设计模型的部件图

在实现阶段的过程中，可以选取下列图的说明来辅助编程。

- ☐ 类的规格说明：每个类的规格说明详细显示了必要的属性和操作。
- ☐ 类图：显示类的静态结构和类之间的关系。
- ☐ 状态图：显示类的对象可能的状态、所需处理的转移以及触发这些转移的操作。
- ☐ 包含某个类的对象的动态图（顺序图、合作图、活动图）：显示该类的某个方法的实现或别的对象是如何使用该类的对象的。
- ☐ 用例图和规格说明：显示系统需求和结果。

编码期间也可能会发现设计模型的缺陷。这时需要开发者修改设计模型。修改设计模型时一定要保持设计模型与编码的一致性，以便将来易于维护。

5. 测试和配置

完成系统编码后，需要对系统进行测试，它通常包括：单元测试、集成测试、系统测试和验收测试。在单元测试中使用类图和类的规格说明，对单独的类或一组类进行测试；在集成测试中，使用组件图和合作图，对各组件的合作情况进行测试；在系统测试中，使用用例图，以检验所开发的系统是否满足用例图所描述的需求。

系统的配置是实际的交付系统，包括文档和组成模型等。对本章提到的集中监控系统而言，它是一个典型的客户/服务器系统。可以用配置图显示系统的物理结构，如图 11.26 所示。从表面上看，配置图能显示系统设备之间的关系以及显示节点跟可执行软件单元的对应关系。然而一旦某个节点内部的对象或可执行部件过多（超过 5 个），就很难完全用配置图清楚描述这种关系。

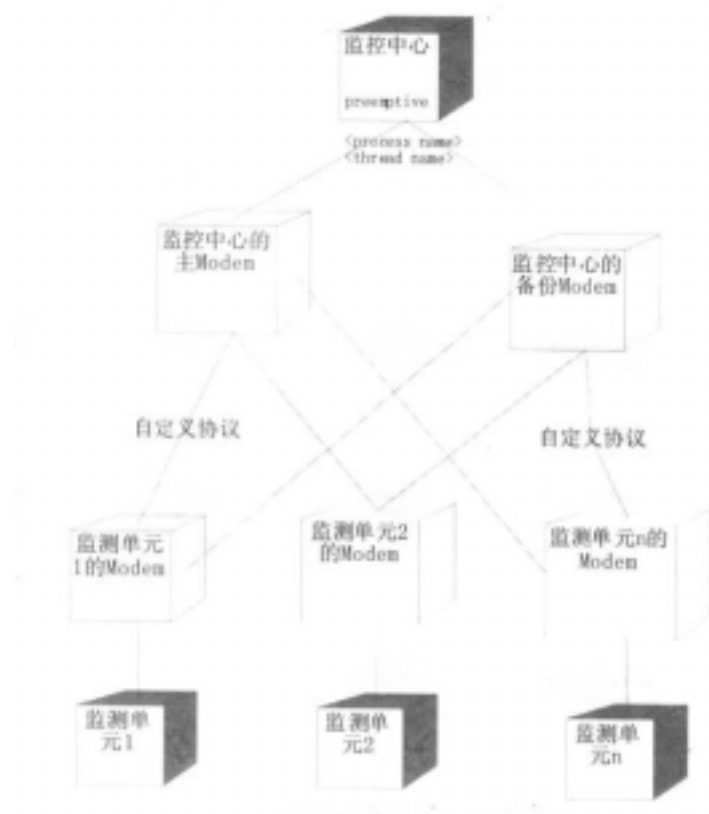


图 11.26 配置图

6. 总结

本章所举的集中监控系统使用 UML 建模。其中首先要掌握的是如何使用用例技术正确描述系统需求。UML 中的类图描述的是系统中类的静态关系，对象图有助于对复杂类的理解。在系统开发过程中，类图可应用于分析、设计和实现阶段。类的包化有助于进行系统结构设计。

UML 的动态模型包括状态图、顺序图、合作图以及活动图。在集中监控系统中，顺序图和用例图对描述各对象的交互非常有用，是系统分析、设计和实现阶段最重要的支持手段之一。

总之，UML 提供的九种视图从不同应用层次和不同角度为系统从系统分析、设计直到实现提供有力支持。在不同的阶段建立不同的模型，建模的目的也各不相同。

UML 为用户建模提供了强大的支持，并提供了很大的自由度。用户在遵循增量迭代开发的原则下，完全可以根据自己所开发系统的特点，在每次迭代的微过程（分析、设计、实现、测试和配置）中，灵活地选用 UML 所提供的各种图。

11.5 参考

下面给出了 Delphi 源代码和模型元素映射关系。

(1) Delphi 源代码项为:

```
type SampleClass1 = class
  (BaseClass, SampleInterface)
{...}
end;
```

type SampleClass1 模型元素如图 11.27 所示。

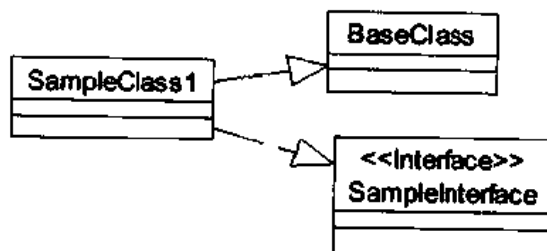


图 11.27 type SampleClass1 模型元素

(2) Delphi 源代码项为:

```
type SampleClass2 = class
private
  AnAttribute : Integer;
  ARole : SupplierClass1;
public
  procedure AnOperation
    (arg : Integer) ;
  property AProperty : Integer
  index 2
  read AnAttribute
  write AnOperation;
end;
```

type SampleClass2 模型元素如图 11.28 所示。

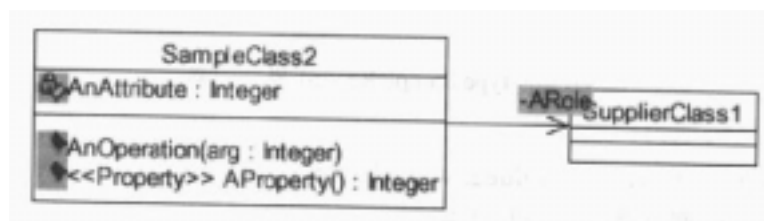


图 11.28 type SampleClass2 模型元素

特性指示器 (Property specifiers) 存在相关的 Rose 操作的代码生成特性。

(3) Delphi 源代码项为:

```
type SampleClass3 = class
private
  ArrayRole1 : array of SupplierClass2;
  ArrayRole2 : array [1..10] of
    SupplierClass5;
  ArrayRole3 : array [SampleRange] of
    SupplierClass3;
  ArrayRole4 : TItems;
end;
```

type SampleClass3 模型元素如图 11.29 所示。

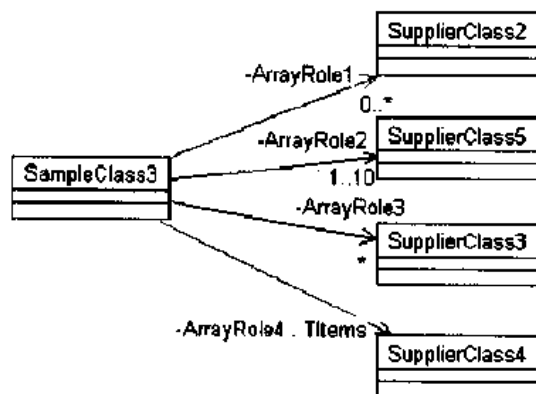


图 11.29 type SampleClass3 模型元素

(4) Delphi 源代码项为:

```
type SampleRecord = record
  attr1 : Integer;
  attr2 : Real;
end;
```

type SampleRecord 模型元素如图 11.30 所示。

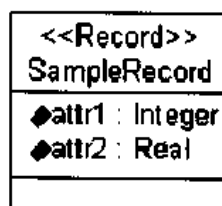


图 11.30 type SampleRecord 模型元素

(5) Delphi 源代码项为:

```
type SampleEnum = ( value1, value2, value3 );
```

type SampleEnum 模型元素如图 11.31 所示。

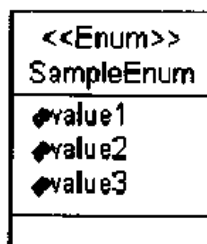


图 11.31 type SampleEnum 模型元素

(6) Delphi 源代码项为:

```
type ProcedureTypeSample = function ( arg :
    Integer ) : Integer;
```

type ProcedureTypeSample 模型元素如图 11.32 所示。

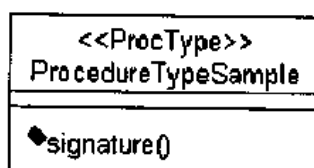


图 11.32 type ProcedureTypeSample 模型元素

过程类型用 ProcType 模板 (Stereotype) 映射到 Rose 类。

(7) Delphi 源代码项为:

```
type SampleRange = 1..10;
```

type SampleRange 模型元素如图 11.33 所示。

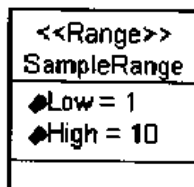


图 11.33 type SampleRange 模型元素

子范围在 Rose 类中表示为 “Range” 模板。

(8) Delphi 源代码项为:

```
type TOldType = class
end;
```

```
type SampleTypeId1 = TOldType;
```

type TOldType 模型元素如图 11.34 所示。

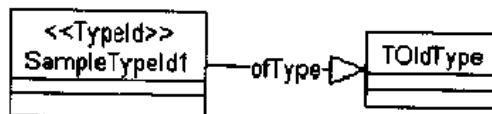


图 11.34 type TOldType 模型元素

一个类型 id 表示为 Rose 类的“TypeId”模板。如果重命名的类是 Delphi 的标准类（比如 Integer），属性用来建模这个类。否则，使用继承关系。

(9) Delphi 源代码项为：

```
type SampleTypeId2 = Integer;
```

type SampleTypeId2 模型元素如图 11.35 所示。

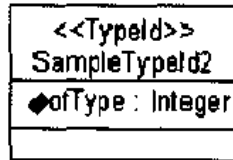


图 11.35 type SampleTypeId2 模型元素

(10) Delphi 源代码项为：

```
type SampleClassRef = class of TObject;
```

```
type SamplePointer = ^TOtherType;
```

```
type SampleFile = file of Integer;
```

```
type SampleArray = array of array of Integer;
```

Rose 类模型元素如图 11.36 所示。

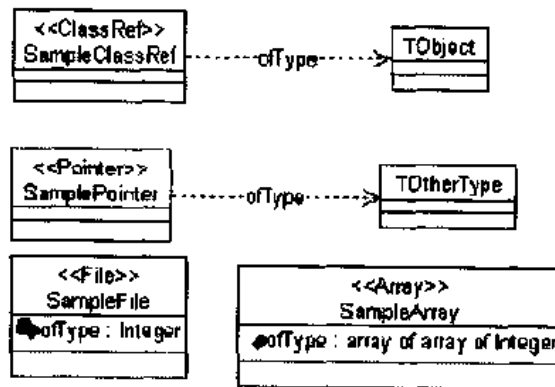


图 11.36 Rose 类模型元素

Delphi 的指针类型、类引用、数组类型、集合类型和文件类型通过使用模板来表示为 Rose 类。如果 Delphi 类型引用类其他用户定义的类型，则以从属 (Dependency) 表示。如果 Delphi 类型引用一个基本的 Delphi 类型 (Integer) 或者一个复杂的类型 (如嵌套数组)，则引用建模为一个属性。

(11) Delphi 源代码项为：

```
Unit SampleUnit;
```

```
interface
```

```
type SampleClass1 = class { ... } end;
```

```
implementation
```

```
end;
```

Unit SampleUnit 模型元素如图 11.37 所示。

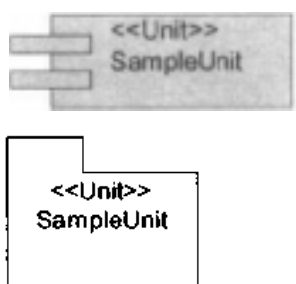


图 11.37 Unit SampleUnit 模型元素

单元映射到一个组件和类别。它的类为组件指派的类。

本章小结

Rational Rose 是 Rational 全面解决方案的一个软件模块，为项目提供可视化建模，适合于管理项目开发的需求分析、概要设计、详细设计。

本章首先介绍 Rose 的基本情况，Rose Delphi Link 在 Delphi 项目中的应用，然后介绍 Rose 支持的 UML 的基本元素，UML 在项目不同的阶段作用，最后给出 Rational Rose 在项目设计中的具体应用。

为更好地了解 UML 语言，建议读者阅读北京希望电子出版社的《时代新潮流 UML Programming Guide 设计核心技术》和清华大学出版社的《Practical Object-Oriented Design with UML》。关于软件工程的书，建议读者阅读机械工业出版社的《Software Engineering A Practitioner's Approach》和《软件需求》两本书。关于 Rose，请读者参考 Rose 提供的帮助文档。



Powered by xiaoguo's publishing studio
QQ:8204136